AFOSR - TR - 75 - 1679

8

FG

ADA019571

Machine-independent Generation of Optimal Local Code

Joseph M. Newcomer

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

See Form 1473

Machine-independent Generation of Optimal Local Code

Joseph M. Newcomer

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania, 15213

May 16, 1975

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense under contract number F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

Acknowledgements

I find it difficult to properly acknowledge all the people who are responsible for this thesis. The guidance and opportunities made available to me over the years since I first encountered the field of computing has been, I find upon reflection, enormous. To attempt to list everyone would be futile. To all those who have helped me in the past: parents, relations, professors, employers, co-workers; I am grateful.

There are, of course, those whose help has been most immediate in the formulation of this thesis. I appreciate the criticisms and observations by the members of my committee: Professor Mary Shaw, Professor Jack Buchanan, and Professor Charles Kriebel. I am especially in debt to my adviser, Professor William Wulf, who not only provided technical discussion and many insights into the problem which this thesis treats, but also provided invaluable encouragement when it was most needed.

Abstract

There has been extensive research into the automatic generation of compilers. Much of this has concentrated on the issues of syntax and semantics, while little has been done on the problems of code generation. This thesis represents one approach to the latter problem. A model of a compiler-compiler is presented, with the research focussing on the construction of one component of the compiler, that module which determines the possible code sequences which realize a given program. The input to this component is a set of code sequences which are possible realizations of each language construct. This thesis concentrates on the automatic generation of these code sequences from a formal description of the hardware and the language. A notation is developed for representing machine instructions, and a prototype system has been constructed to demonstrate that this notation is amenable to automated analysis.

Table of Contents

Chapter I

Introduction

Background

Problem statement

Two long-standing problems, the construction of compilers and the transfer of programs between machines (i.e., portability), have recently been emphasized by the current proliferation of machines and languages today. This thesis addresses certain aspects of these problems with respect to generating machine code sequences from higher-level language constructs.

Given a specific language and a specific machine, the construction of a good compiler (and a correct compiler) may take two or three years. In a research environment the impact of this long time frame is felt in many ways. It is difficult to explore new areas of language design; rapid turnover of personnel (such as undergraduate and graduate programmers) makes it difficult to maintain continuity in a project; other research projects which need to develop or use a specialized language may be forced to choose other, perhaps less desirable, alternatives. In the commercial environment the impact is more direct. Each new computer must be produced with a complete complement of programming languages in order to be a marketable commodity. A delay of six months could well alter the market position of a new computer.

The issue of portability is particularly important in research environments, and is of considerable concern to the very large class of end users of programs. Ideally a program is a realization of an algorithm. The user is generally interested only in the algorithm and not in the program which represents it. Such occurrences as the replacement of one machine (or operating system) by another, for whatever reason, should be of little concern as long as the programs continue to run. Unfortunately, for reasons we shall discuss later, it is rarely the case that programs can be transferred directly from one operating environment to another.

In many cases the end user is only concerned with the results: if the program is transferred to the new environment and produces correct output then the user is satisfied. The isolation of the user from the environment means that issues of efficiency often are ignored; hence the incidence of 370/145's simulating 1401's simulating 650's simulating tab card equipment. It is when the user must pay real money from a finite budget or spend time waiting for a timesharing system to respond that issues of efficiency become important. Computer cycles are a finite resource. Any cycles wasted because of inefficient code are not available for other purposes. As long as we must live with a finite resource, we should optimize its utilization as much as possible.

This thesis is concerned with one aspect of a particular approach to the solution to these problems, namely compiler-compilers or translator writing systems. Compiler-compilers are programs which, when supplied with appropriate information, will generate a compiler for a specific language and machine combination. They have been studied for many years and have had some limited success in language research environments. They have not, however, had much success in the commercial environment. There are two related reasons for this. First, compiler-compilers have tended to produce compilers which, in turn, produce object programs of poor efficiency. Second, they have tended to focus on the issues of lexical and syntax analysis to the exclusion of code generation. As a consequence, much work remains for the user of the "automatic" systems to complete; and that which is related to code generation is usually tedious, complex, and subject to error.

The specific concern of this thesis is to investigate a method to be employed in the construction of compiler-compilers such that (1) the resulting compiler produces code comparable to that produced by the best optimizing compilers, and (2) to do so in a context that makes the specification of code generation relatively easy. An outgrowth of this research is the ability to specify the machine characteristics of the target machine, making it possible to construct compilers which can produce code for many different machines.

It is not an objective of this thesis to produce a compiler-compiler. However, the research has developed a program which, given appropriate information about a language and machine, will supply information to a hypothetical compiler-compiler.

History

The problems of compiler construction and machine portability have been investigated for many years. In order to view this research in proper perspective it is necessary to examine some of the history of these efforts.

The need for machine independence has long been realized. The underlying philosophy is that we wish to reduce as much as possible the difference between algorithms and their realizations in programs.

There are many issues involved in transferring a program from one machine to another, both in translation and execution.

The translation problem is simply that of converting some representation of the program to a form which can be operated upon by the target machine. This means that there must be a translator which accepts the source program and outputs some representation of the object program. Unless the language is one with widespread acceptance, such as FORTRAN, COBOL, or BASIC, this step alone poses the most serious problem. Also, even if a translator exists, it must accept all the features of the source language and produce correct code for them. Quite often the lack of standard notation for the source language or standard syntax for certain classes of operations becomes a virtually impenetrable barrier. One need only look at the dozens of variations of reserved word and input-output syntax represented by the many implementations of Algol (which was intended to be an international standard) to realize the difficulty this imposes.

Given that a program can be translated, it is then necessary to actually execute it. Most languages require the existence of an execution-time environment ("run-time-system") to perform everything from data space allocation to input-output. It is not only necessary to specify the syntax and semantics of the computational and control statements of the language but also to specify the semantics of the execution time environment (including input-output) as well. Even those languages with fairly rigorous language specifications often leave the semantics of the execution environment either unspecified, or worse, so ambiguously specified, that the variations in implementation makes the transfer of programs between any two environments nearly impossible. In particular, the semantics of input-output is usually determined by the behavior of the operating system under which the program will run, rather than forcing the operating system to provide a set of facilities satisfying a standard semantics[1].

Even if all of these problems were solved, however, we still would have the problems of the physical hardware representation. These include such issues as the character set and collating sequence, word length, floating point precision, and behavior under exceptional conditions, e.g., does fixed overflow generate an interrupt or set a flag? If we can avoid all of these problems then we have, in fact, achieved a portable program. For a discussion of these issues and some responses, see Warren [War74].

If we assume certain standards of external representation (e.g., the character set and the formal syntax), we are then left with issues such as machine word length and radix. Programs which pack data into words usually presume a certain radix or word size. Floating point computations assume (or tend to assume) certain properties of the floating-point arithmetic unit, such as the base, radix, rounding, etc. as well as significant factors such as the largest or smallest number which can be represented.

An interesting approach to the latter problem is described by Malcolm [Mal72] in the form of some algorithms (expressed as FORTRAN subroutines) which determine the properties of the floating-point unit for any given machine. This defers the binding time of these properties to execution time rather than forcing the binding at compile time (and hence at coding time, in most cases).

Newer languages, such as Algol-68 [vW69] respond to this issue by requiring a

1 A possible cause of this difficulty is the fact that presumably rigorous specifications are made in English prose, a notation peculiarly subject to ambiguity. Alternative notations, such as the Vienna Description Language [Weg72] or van Wijngaarden grammars [vW69] would be more suitable. The use of an informal but rigorous notation for the behavior of the run-time support system, such as the notation suggested by Parnas [Par72], would be a major advance for most languages.

"standard prelude" in which important information such as the largest possible integer or the smallest floating point number is made available. Each compiler defines a standard prelude for its target machine. Unfortunately, both the Malcolm solution and the Algol-68 solution require that the programmer utilize the information made available. Failure to do so will probably result in a non-portable program.

Since we have abstracted the language and machine issues to some degree, it should now be evident that we should be able to achieve machine portability (modulo the issues given on page 4).

In the following five sections we will examine some approaches to the issues of compiler construction and code generation. They are

(1) Compiler-compilers
(2) Code generators
(3) Standardized languages
(4) UNCOL

Compiler-compilers

One might imagine that the problem of machine independence could be solved by constructing a compiler which would compile "any" source language for "any" machine. The word "any" is quoted here since each researcher in the field restricts the classes of languages and machines to those for which he can (or intends to) find a solution. The diversity of languages and machines makes it nearly impossible to consider a solution which is truly universal.

A much simpler solution is to construct a program which accepts the specifications of a language and the specifications of a machine, and then generates a compiler which compiles source programs in that language to code for that machine. Thus we enter the world of compiler-compilers. The classical compiler-compiler paradigm is shown in Figure 1.

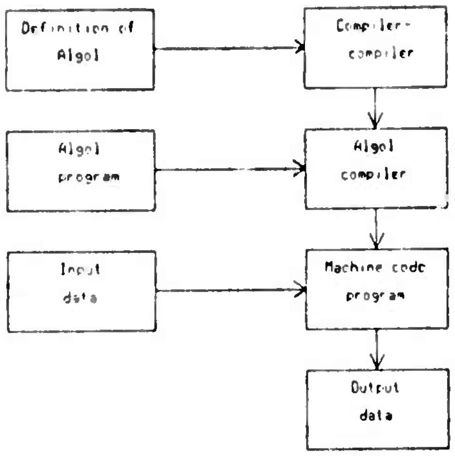

Figure 1: The Classical Compiler-compiler structure (after [Hop69])

In the past there have been many attempts to construct compiler-compilers [Ev64, Fel64, Ros67, WaS67, CLE69]. A classic paper by Feldman and Gries [FG68] reviewed the state of the art in 1968.

The system described by Evans [Ev64] generated a syntax analyzer from an input specification of Floyd productions (the particular adaptation became known as Floyd-Evans productions). The input to the parser was a stream of lexemes produced by a lexical analyzer ("subscan") and the output was another (linear) stream of lexemes which were used by the code generator. The use of Floyd-Evans productions was extended by Feldman [Fel64] in FSL, which did not output a lexeme stream, but instead invoked a "semantic routine" for each successful production. The semantic routines were written in FSL, and allowed the compiler writer to specify the semantics of the compiled code without specifying the actual form of the code. The ideas of FSL have been extended by others, notably by White in JOSSLE [Wh73].

The general-purpose table-driven compiler described by Warshall and Shapiro [WaS67] allowed specification of the syntax of the language by a slightly modified BNF (context free) grammar. The output of the syntax analyzer was a tree representation of the source program. A second notation allowed the specification of matches against the tree, which could output "macros", machine-independent representations of the operations to be performed. The expansion of these macros into machine code for the target machine is accomplished by the use of a third notation; the output from this phase is an assembler program which can be translated by conventional means.

The Brooker-Morris system described in [Ros67] offers elaborate syntax analysis specifications. It is interesting in that both the syntax specification and the output ("format") specifications are expressed in a single notation. It is claimed that the notation allows for the handling of block structured declarations and typed data objects within the syntax specification.

TREEMETA [CLE69] is a more unified implementation of the same ideas. It allows the compiler writer to specify the syntax of the language in an extended BNF grammar. The output of the parser is a tree with labeled nodes, where each label is the name of an "unparse rule". An unparse rule specifies a series of tests (matches) to be performed upon the node, and if a test succeeds it specifies a set of output rules which send text to an output file. The output text may then be passed through a conventional translator (such as an assembler) to obtain the object program.

Code Generators

One of the major difficulties in constructing a compiler is the generation of machine code. This is especially important if the language is considered machine-independent, and intended to run on a variety of machines. Ideally, one wishes only to specify the characteristics of the machine, and have a system which automatically produces a code generator tailored to the specific machine. Such systems are discussed by Miller [Mil71] and Donegan [Don73].

Miller describes a system, DMACS, which takes a description of a machine in terms of registers, storage, and the allowable operations which map between them (including such concepts as data accessing functions). He then specifies a method of generating code using the information as a data base. The input to the code generator consists of simple two-address code, with an implicit "result address" for each line of two-address code. For example, [Mil71, p 17]:

Line	Op	Operands
1	MUL	C,D
2	ADD	1,B
3	ASSG	A,2

represents the compilation of "A = B + C * D". The first line multiplies C and D; the second line adds to the result of line 1 the contents of B; the third line assigns to A the result of line 2.

An important idea here is that the compiler need only generate the indicated code sequence. It does not matter if C is an integer, D is a real, B is a bit field within a word, or even across word boundaries, and A is a register explicitly designated by the user. The DMACS system will provide the necessary mappings, allocate space for the intermediate results, and provide the necessary access functions to obtain the operands and store the results. Thus the front end of the compiler is constant across all machines, and the code generator discovers the code sequences based upon a description of the machine. Significantly, the code generator can discover if the operands given are suitable for the operation (e.g., MUL) and convert them as required (e.g., move C to a register, convert to floating point, perform floating point multiply).

The model presented by Donegan [Don73] is more elaborate, and is closer to the compiler-compiler idea. A preprocessor accepts a notation describing the translation of the intermediate representation of the program into machine code. The output of this preprocessor is then compiled to produce the code generator. Code generation is considered as a finite-state machine operation, where an accepting state emits an instruction. Given any state, there are a set of possible transitions derived from the machine description. The preprocessor examines the possible transitions and produces a code generator which will choose the minimum-cost transition.

Standardized languages

An alternative to the compiler-compiler approach is to define a standard of syntax and semantics for a language, such that the language can be implemented on a large class of machines. If this approach were completely successful it would be possible to transfer most programs directly from one system to another, providing direct source language portability. This has been attempted for several languages; FORTRAN, COBOL, APL, and PL/1 are the first examples which come to mind. If there were any adherence to the standards specified for these languages, indeed we would have a high degree of machine portability; in fact, we do not. One of the problems with such complex languages is the lack of completeness of the compiler (there are authorized subsets in many cases) and the number of extensions to the compiler which the programmer has taken advantage of (nearly every compiler has nonstandard extensions, either deliberate or accidental). However, if we choose a language of suitably restricted syntax and semantics it is possible to transfer programs written in it to another machine.

A variant of this approach has been taken by Bell Laboratories where they have defined a "standard" subset of FORTRAN IV. This is a restricted subset of the language which they have found empirically to transfer to a large number of machines and remain compatible, syntactically and semantically, with each of their FORTRAN IV compilers[1].

The inverse of the Bell Laboratories' approach has been taken by the Department of Defense (DOD) with respect to the COBOL language. A distribution tape contains a set of COBOL programs with rigidly specified behavior [Bai72]. In order for a COBOL compiler to meet the standard for a certain subset of the language, it must successfully compile and execute a certain subset of these programs.

Macro systems

For some applications it is possible to use a language of very restricted syntax and very rigid semantics to specify a program. The translation problem can then be viewed as mapping each statement in the language into a set of machine instructions which produce the desired effect. This translation can usually be performed by a macro processor of some sort, in which the components of each statement (the actual parameters) are substituted for placeholders in a template (the formal parameters in the macro body). The result of this substitution is the text for a language translator (compiler or assembler). When this text is translated the result is a machine-code program which produces the desired output.

One of the earliest uses of a macro system to specify the code templates for a high-level language was in the MAD language [AGG69]. In this system, a special-purpose macro processor was included in the MAD compiler. New data types could be defined, and the operations on these new types could be defined by specifying the

1 To validate programs for "portability" they have a program which reads a FORTRAN IV program and verifies that it uses only the portable subset of the language [Ryd72].

actual machine code to be used. A template of this form would be expanded whenever the operator it defined encountered the data types it accepted; in this manner standard operators (such as "+") could be defined over new data types (such as complex numbers). The formal parameters in the template were replaced by actual parameters during the expansion. There were also conditional compilation facilities which controlled the text expansion.

Macro systems are attractive from the viewpoint of portability; ideally, the program text, except for minor character set differences, can be read by nearly any machine. Once an algorithm is expressed in terms of macro text which can be expanded, it is in theory possible to transfer it to any machine.

There are several languages which are specified by macro language definitions; the two cited here were chosen because they are both well-documented in the literature and successful enough in practice to be more than theoretical approaches.

One of the more successful attempts at machine portability is the Mobile Programming System of Orgass and Waite [OW69, Wai70]. A bootstrap macro processor called SIMCMP is written in a subset of FORTRAN IV, about 110 lines of code, and trivially translatable by hand to nearly any other language. It is used to compile a more powerful macro processor called STAGE-2. STAGE-2 is written in a language called FLUB, which has a very limited syntax and primitive semantics, and can be translated by SIMCMP. Once STAGE-2 is running, other systems may be written in the more powerful macro language which it translates. With proper care in the specification of the semantics of these macros any program written for translation by STAGE-2 can be moved to another machine with a minimum of effort.

Another very successful approach along these lines, for a single application, is the SNOBOL Implementation Language (SIL) designed by Griswold and his associates for implementing SNOBOL-4 [Gris72]. SIL consists of about 130 macros oriented explicitly towards the implementation of SNOBOL data structures and internal algorithms. It was also designed to obey the syntactic restrictions of an archetypal assembly language macro processor, so in some ways it represents the intersection of features of several such macro processors. It has been used with great success (although not without difficulty) to implement SNOBOL-4 on at least ten different machines (as of 1972).

UNCOL

There has been an assertion [Str58, Ste61] that if the syntax and semantics of a language are sufficiently rigorous and at the same time not very far removed from a machine representation that (ideally) we could code all of our programs in this language and have them execute on any machine. One obvious defect is that such a language would undoubtedly be too low-level to actually program in. In fact, this is true; but if we had all of our compilers produce code for the machine which "executed" this language, then the output of any compilation could be translated and run on any machine.

An early proposal along these lines was UNCOL [Con58, Str58, Ste61]. This involved creating a single "universal" language into which all other languages could be compiled; it would then be necessary only to write programs which would translate UNCOL into the specific machine languages for the target machines. Had this solution been successful, it would have reduced the m x n problem to an m + n problem (see Figure 2). The m x n problem is characterized by the fact that for m languages and n machines it is necessary to construct (n x m) compilers to be able to run a program written in each language on each machine. Using the UNCOL solution, only m + n "translators" need to be written---one for each source-language-into-UNCOL transformation and one for each UNCOL-into-machine-code transformation.

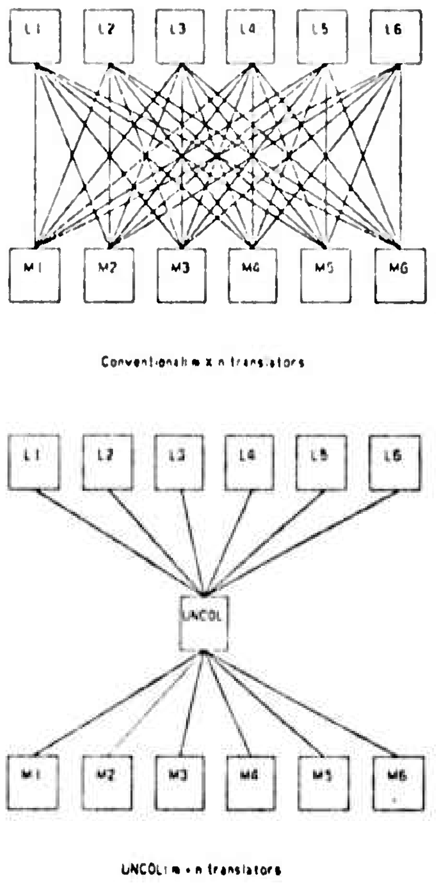

Figure 2: The reduction of the m x n problem.

There have been several attempts at UNCOL-like solutions. In general these have been restricted to defining particular intermediate-level languages for the implementation of specific high-level languages (the 1 x m problem). The SNOBOL Implementation Language can be viewed as the output of a (human) compiler which allows a single language to be implemented on many machines. The OCODE representation of BCPL [Rich71] is produced by the BCPL compiler (which is written in BCPL). Thus to bring up BCPL on any system it is only necessary to translate OCODE into machine language. Once an OCODE translator is built, it is only necessary to translate the OCODE representation of the compiler. The result is now a BCPL compiler which runs on the desired machine. This compiler may then be used to recompile the BCPL compiler source text, or any other BCPL program.

One of the most recent proposals for an UNCOL-like system is JANUS [Cole74], an intermediate textual representation for the output of a compiler. A translator based on the STAGE-2 macro processor [Wai70, see page 12], which is already portable, is then used to translate JANUS into assembler code for the target machine. JANUS is noteworthy because it is not designed for any particular language (but rather a broad class of languages), and it is one of the most complete proposals for a truly "universal" intermediate language.

Critique

Compiler-compilers and their code generators

All of the early major work in compiler-compilers seems to have concentrated on syntax analysis [Ev64, Ros67, WaS67], with little or no work on code production or optimization. In particular, the concepts of global, or machine-independent, optimization have not been handled at all. The work of Geschke [Ges72] has now shown a notation for specifying how to detect and process global optimizations, such as those discussed by Cocke and Schwartz [CS70]. Compiler-compilers will have to incorporate these abilities if they are to compete with so-called "hand-coded" compilers.

Code production in early compiler-compilers was fairly simple, and optimization was usually restricted to keeping track of which results were currently in the accumulator[1]. More sophisticated code production and optimization

[1] The fact that most of this work was done on single-accumulator machines made the more sophisticated issues of register allocation irrelevant.

was usually performed by hand-coded assembler routines rather than by using some notation at the compiler-compiler level, as in [Ev64]. In this system, the compiler-compiler provided a notation for specifying the translation of a source language (Algol-60) into a linear stream of tokens. Typical semantic problems such as requiring that an identifier be declared before use, declared only once at a given block level, used consistently with its declaration, etc., as well as issues about coercion between integers and reals, were handled by a second, hand-coded phase of the compiler. This approach was not as successful as one would have hoped. The second phase of the translator was sensitive to the form of the token stream. Although the syntax could be changed easily, any change in the order of the tokens, or in their type, required modification of the complex second phase in order to ensure that the mapping into machine code would operate correctly, or at all.

One of the lessons learned from the Algol compiler of Evans [Ev64] was that syntax analysis was one of the easiest parts of the translation process (a view not generally held by the computing community at that time), and that semantic analysis and code generation were more difficult and more important problems.

Later work on compiler-compilers was influenced by the recognition that syntax analysis is not the major issue, and the effect of this influence was the provision of capabilities such as being able to specify the mapping from the semantics of the source program to the semantics of the target machine [Fel64, Wh73, Don73, Mil71, CLE69]. There are several shortcomings in these systems. The most serious is that they are either too far from, or too close to, the actual machine representation of the object program to make machine independence simple to handle and at the same time maintain the goal of efficient object code. Abstract semantics such as those embodied in FSL [Fel64] or JOSSLE [Wh73] adequately abstract storage allocation, forward references, and other concepts independent of efficient code, but they specify machine operations such as addition by simply requesting the production of code to evaluate the "+" operator (the "code brackets" of FSL). In any given machine there may be a score or more different ways of implementing this language operator, each having particular cost tradeoffs[1]. The exact code sequence required to produce the desired effect is left to the ingenuity of the person coding the semantic routine.

TREEMETA provides a convenient representation for specifying the syntax of a language but ties the semantics and code generation so tightly to the idea of the treewalk that converting a program to produce efficient local code on a different machine can involve rewriting the entire code production phase. The internal representation is inherently a tree; the use of a directed acyclic graph (dag) to represent common subexpressions is not possible in standard TREEMETA. Also, in standard TREEMETA it is impossible to restructure the tree once it has been constructed, so that "code motion" optimizations (such as moving constant computations outside of loops [CS70, Ges72]) cannot be performed.

The work of Miller [Mil71] and Donegan [Don73] provide convenient notations for specifying the case analysis required to produce code from an intermediate representation. However, optimum code production requires careful case analysis based on the actual (often pathological) behavior of a machine. The more the compiler

1 Note that addition can be accomplished by "ADD" instructions, "INCREMENT" instructions, indexing, effective address calculation, and all these same variations with use of a "SUBTRACT" instruction! The result is that the BLISS/11 compiler must actually consider 32,000 possible cases of operand evaluation for a single "+" node (although, in fact, most of these cases are redundant).

attempts to exploit the behavior of the particular machine the more likely it is that it will run afoul of these quirks. This can be attributed to either lack of adequate documentation of the machine or the inability of the human mind to cope with the huge number of variables that seem to be involved.

In the classical model of code generation, register allocation and code production are joint operations, where the code generator obtains registers upon request. This allocation strategy ignores many issues of global register optimization, and usually only works well in the absence of common subexpressions. There are other problems caused by lack of global knowledge, such as optimal selection of the intermediate result register to prevent unnecessary transfers of data[1]. In an optimal global allocation strategy, the occupancy of a register by a result is a complex function based upon such factors as the required lifetime of the result, the importance of the result, and the need to preserve the result in a register. It is complicated by such issues as requiring the specific register involved for a specific purpose, e.g., parameter passing. In machines with a small number of registers (such as minicomputers) one cannot reserve a register implicitly for such a purpose without degrading the quality of the code produced. In the model of the compilation process which we use (given in section 2), the classical code production phase is divided into several components, of which register allocation is only an intermediate step. Finite-state models such as those assumed by classical code generators are inadequate to model this style of compiler construction.

Donegan makes the observation that even when the case analysis required for code generation is done manually, and a case table produced (readable by humans),

1 Known in BLISS/11 as "targeting".

when this table is translated to code in most currently available notation for code generators the meaning is almost entirely lost (e.g., as in [ER70]). One of the strengths of his system is that once the case analysis has been performed, the translation into a suitable notation is relatively straightforward, and the actual production of a code generator is automatic.

The major weakness in the classical finite-state machine model of code generation, as exemplified in nearly all code generators associated with compiler-compilers, is that there is insufficient information available at any point to produce really optimal code. The three-address code model (or n-address, where n≥3) is one of the most difficult intermediate representations to optimize, although it is one of the most common representations used. However, a collection of techniques for the n-address model is described by Frailey [Fra70].

Macro systems

With only a few notable exceptions, the use of a macro language to transfer programs from one environment to another has not met with much success. The class of macro languages as exemplified in the MAD definitional facility are clearly machine and environment dependent although they constitute a valid approach to language extension within an environment. In most cases, the available macro processors are not compatible, and one is forced to various artifices to achieve a successful transfer. The syntax of the language must be restricted to that which can be accepted by the macro processor; the syntax and semantics can also be restricted by the power of the macro processor, such as conditional text inclusion, ability to omit parameters, and ability to store and manipulate global state information.

The macro system of Orgass and Waite [OW69] avoids many of these problems simply by defining the macro processor in such a way that it can be easily bootstrapped onto another machine. This approach is one of the prime reasons for its success. The SNOBOL experience [Gris72] attempted to avoid some of the difficulties by using a syntax which represented the intersection of several known assembler macro processors. The difficulties encountered in transferring the system from one machine to another were due, in part, to the incompatibilities between the SIL definition and the abilities of the macro processor. In terms of coding the SIL implementation, it was often necessary to place undue restrictions on the syntax of the language simply because some macro processors could not accept a more desirable syntax. This seems to indicate that unless great care is exercised in the design of a macro implementation that it would be no easier to transfer a set of macros from one machine to another than it would be to transfer any other type or program.

Macro systems also present severe problems with respect to code optimization. Global state information which can be made available to an optimizer under other representations (such as trees or graphs) is lost when the program representation is processed by a macro processor, due to the single-pass nature of such systems. It should be pointed out, however, that Waite has illustrated a very simple local code optimizer implemented entirely within the STAGE-2 macro processor [Wai69]. The optimizations are very similar to those of conventional code generators in compiler-compiler systems, such as being able to detect that the result in an accumulator is used in a following computation. Properties such as commutativity are also used to advantage. However, such a system is still not powerful enough to detect more global optimizations.

UNCOL

The UNCOL solution suffered from several basic problems. In particular, the UNCOL representation had to embody the union of the semantics of all languages past and future. To do this would require a single representation able to express the low-level semantics of FORTRAN, Algol, PL/1, SNOBOL4, APL, LEAP [FR69], Algol-68[vW69], and SIMULA-67 [Dah67] (to name a diverse set). It is not likely that any language could in fact represent this diversity of semantics and still remain manageable, and simple or efficient to compile.

In addition, UNCOL was supposed to be a very low level machine code for an abstract machine[1]; the translation to an actual set of machine instructions was supposed to be very simple. It is not likely that such a low level representation could be efficiently translated across the variety of architectures currently available, even ignoring such radical departures from "conventional" architecture as ILLIAC-IV and STAR. A representation which would be efficient for a 7090 (three index registers, one accumulator) would undoubtedly be extremely inefficient on a IBM/360 architecture (16 fixed-point accumulators/index registers, 4 floating-point accumulators) compared to code generated for that architecture. Since our goal is to produce <u>efficient</u> machine code we must find such a solution unsatisfactory.

Note that this rejection of the UNCOL-class of solutions is based upon a stated goal of efficient object code. The validity of this solution is not questioned for those cases where portability is considered a more important goal, although we expect that the long-term results of our research will make it possible to achieve both portability and efficiency.

1 A one-address machine without an explicit accumulator [Ste61].

If we treat the SNOBOL Implementation Language (SIL) as an UNCOL-like notation (although admittedly for only one source language), we find that it produces a system about three times larger and three times slower than an equivalent non-portable system (SITBOL, see [Gim74]). SITBOL is, however, not portable outside the environment of a PDP-10, and users who wish to produce portable SNOBOL4 programs must only use the SNOBOL4-compatible subset of SITBOL.

The conclusion here is that with current techniques one tends to sacrifice speed and size for portability; not just by a few percent, but by large factors. These factors are especially significant if one considers the large community of minicomputer users; the machines possess both small address spaces and small physical memory. A factor of two in the size of a program is extremely important here. A system which preserves machine independence at the cost of physical size may not actually be portable if programs cannot operate in a large class of real machines.

Standard languages

Another approach is to define an "abstract machine" in terms of some high-level language, i.e. an Algol, FORTRAN, PL/1, or APL "machine", and then provide a mapping from this to a real machine. This approach has suffered from the lack of a notation for the specification of the semantics of such languages, ambiguities in the specifications, and/or errors of commission and omission in the implementations. This is further complicated by extensions which each compiler embodies which are not part of the standard. Note that although the previously listed languages in fact have "standard" definitions, it is rare that a complex program operating in one environment can in fact be transferred to another environment, independent of issues of machine word length, floating point precision, or radix of internal representation. Again, we note that

SNOBOL4 as implemented in SIL avoids many of these problems by defining a standard implementation, not just a standard definition.

Problems with optimizations

Nearly all optimizing compilers embody certain assumptions about the data upon which they operate. For example, nearly all such compilers make the assumption that "$A \equiv -(-A)$" is true, and thus collapse unary minus operations [Fra70, KKR65]. This equivalence is only valid as long as one assumes that no variable takes on the largest possible negative value when the hardware uses two's complement representation[1]. Compilers which use this equivalence assume that the largest negative number is an "unlikely" occurrence. On the other hand, they will naturally assume that the value 0 is a common occurrence, and thus not use the equivalence "$(A/(B/C)) \equiv ((A*C)/B)$", even though floating point multiply is often substantially faster than floating point divide. (Note that we ignore for a moment the issue of floating point accuracy of the result).

Beatty [Bea72] refers to the former as a use of an axiom in its "permissive role", and the latter as the use of an axiom in its "strict role". The decision of whether to interpret an axiom in its strict or permissive role seems to be based only upon a stochastic model of values, which probably bears no resemblance to the actual values encountered in a given application. In particular, there are a large number of axioms which, when used in their permissive roles, can be employed if the range and accuracy of the data is known, but which must be rejected in the general case. This leads to the desire to be able to specify these to the compiler.

1 In two's complement the largest possible negative number is -2^n (for n bits of representation) while the largest positive number is 2^n-1. The negation of the largest possible negative number, which we denote as $-\infty$, does not exist.

Certain well-known "good" programming practices can also result in poor object code. For example, modification of the program is easier if certain knowledge is localized rather than distributed throughout the program. One example is the specification of indices to arrays of data. In languages such as FORTRAN one stores structured data in several n-dimensional arrays, where the indices of certain data are fixed. Thus A(I,1) would be a particular field of the data stored in array A, A(I,2) another field, etc. The only way to localize this knowledge is to associate it with a symbolic name in one (and only one) place and thereafter use only the symbolic name. There is no way to do this in FORTRAN except by the DATA statement or dynamic assignment, both of which leave the assumption that the value can be changed, when in fact this (should) never occur. Even very good compilers such as H-level FORTRAN on the IBM/360 can be made to reject the assumption that a certain variable has one and only one value. Admittedly this is one of the many major defects in FORTRAN as a language, but it shows how a simple lack of knowledge about the data can profoundly influence the efficiency of the object program.

The conclusion from such behavior is that it would be desirable for the compiler to know something about the data on which the program will operate. In one mode of use, this knowledge would allow the compiler to generate checks that the data is within the assumed bounds. This would make it possible to detect errors caused by invalid assumptions or incorrect data. From our viewpoint, it also makes it possible for the compiler to select optimizations which would normally be considered "unsafe" (such as COMMON data prohibiting certain optimizations in FORTRAN), or to avoid optimizations normally considered "safe" (unary minus).

Goals of research

Outline of goals

This research addresses four related issues:

1. Simple, rapid, and inexpensive construction of compilers.
2. Construction of correct compilers.
3. Construction of highly optimizing compilers.
4. Program portability.

As we have seen, there are a large number of techniques for rapidly constructing "front ends" for compilers to handle syntax analysis, consistency checking, and similar issues. There are also efficient techniques for detecting potential global optimizations. The mapping between the intermediate representation and the object code presents the most difficulties, consumes the most time, and is most prone to error. When we add the constraint that the object code must be highly efficient, we increase all of these problems by several orders of magnitude.

We wish to produce a compiler which produces optimum[1] code along some metric (such as time, space, memory accesses, etc.). This should be a real,

[1] There is some controversy about the use of the word "optimum". In fact, we have no analytic lower bound which we can use to judge whether we have achieved optimality, or come within some specified distance of it. One author has suggested the use of the phrase "code amelioration" as an alternative. Painter [Pain70] has suggested a measure of "effectiveness" of optimizations, which is a measure of how well an optimization decreases the cost function over that of the unoptimized version. We will define "optimum" for our purposes as meaning "the best that can be done using all available knowledge about the program structure and machine characteristics". Our goal, then, is to maximize the effectiveness of a compiler by making as much of this knowledge as possible available to it in a usable form.

production compiler to be used for large programs designed to solve complex problems. We are less concerned with the amount of time required to compile the program (within reasonable limits) than with the cost and accuracy of the object code. We wish to produce a compiler which can accept and use knowledge about the data upon which the object program will operate. Thus we will gain in two ways: (1) we will be able to use optimizations based upon known characteristics of the data (e.g., if it will ever be zero); and (2) we can compile checks into the object code to allow the user to debug the program more rapidly by detecting errors previously undetectable.

None of these are particularly new concepts. Most compilers which provide dynamic array index bounds checking have an option to defeat it for some or all arrays (thus asserting that the range of the data will not exceed the limits of the array). Compilers may also allow definitions of constants at compile time, or explicit specification of the exact values which a variable can take (see, for example, PASCAL [Wir71] and MARY [CH74]). We simply intend to exploit these features more thoroughly for code production.

The nature and direction of this research was influenced strongly by the BLISS/11 compiler. Thus it is difficult to explain the exact scope of the research without first presenting the context in which the results must be viewed, the structure of the BLISS/11 compiler.

The choice of the BLISS/11 model was based on three considerations: (1) it is necessary to choose <u>some</u> model of the compilation process; (2) the BLISS/11 model, although not a "natural" decomposition by traditional standards in fact achieves substantially better performance than previous compilers; and (3) the choice of the BLISS/11 model is not restrictive, since it is more general than the conventional compiler model. It should be noted here that the portion of the compiler we are interested in is largely independent of the syntax of BLISS, since it treats only a tree representation of the program. Although a great deal of power is realized because of the GOTO-free nature of BLISS, the optimizations realized because of this are largely independent of the issues we are concerned with.

BLISS/11

The BLISS/11 compiler [Wu71, Wu73] converts programs written in the BLISS language [Wu70] to code for the Digital Equipment Corporation (DEC) PDP-11 [DEC71]. BLISS/11 is a highly optimizing compiler, performing extensive global and local optimization. The compiler itself is partitioned into several phases; a simplified structure is shown in Figure 3. The results of this thesis will be biased toward this structure, as discussed on page 26. A detailed description of the various portions of the compiler may be found in [Wu73]; the description here will be superficial except for those modules of particular interest.

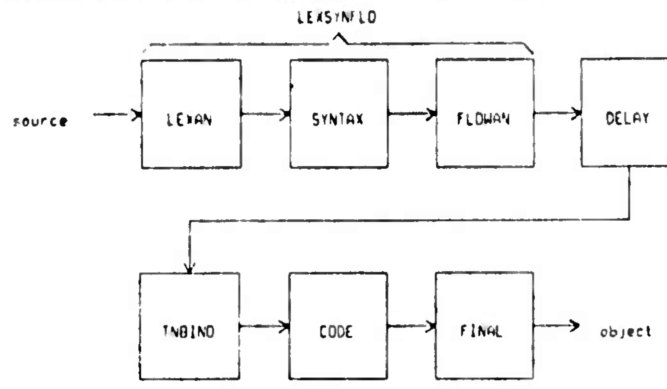

Figure 3: The structure of the BLISS/11 Compiler

Lexical and syntactic analysis are straightforward. Lexical analysis is done by a finite state machine model, and syntax analysis is performed by a recursive descent parsing algorithm. One fact worth noting about the syntax analyzer is that it detects and marks common subexpressions (cse's) "on the fly". Also, it performs nearly all compile-time arithmetic. The resultant tree representation (or more correctly, directed acyclic graph (dag) representation) is then presented to the global optimization detection module, FLOWAN, which is part of LEXSYNFLO. FLOWAN detects all feasible global optimizations, based on the concepts and notations of Geschke [Ges72]. The output of FLOWAN is the tree representation of the program with threads added to link together common subexpressions and indicate the feasible optimizations (e.g., code motions).

The remaining modules, DELAY, TNBIND, CODE and FINAL, perform the functions classically included in a single "code generation" module. The distribution of functions was chosen to maximize the amount of knowledge available when any given function of code generation is carried out.

Briefly, DELAY performs a pass on the tree and "suggests" optimum code sequences. TNBIND performs the resource allocation of registers to abstract registers (called "temporary names"). CODE is a rather straightforward code generator, although it performs more extensive case analysis than most code generators. FINAL performs some sophisticated "peephole" optimizations which are based on code adjacency relationships. Most of the concepts involved in these modules are independent of the syntax and semantics of BLISS.

After we have processed the tree through FLOWAN, the output is presented to DELAY, which in BLISS/11 has the following functions:

> "(1) to determine the "general shape" of the ultimate object code to be produced; (2) to form an estimate of the cost of each program segment; and (3) to determine the evaluation order for the expressions in a program segment" [Wu73].

Note that DELAY does not actually produce any code; it sets flags, computes costs, and indicates desired register usage (desired register usage is not always feasible). This information is used to produce code at a later stage in the processing (the CODE module). DELAY employs a set of heuristics and is based upon a set of assumptions which in fact are very effective, as can be deduced from the performance of the compiler in producing optimized code. However, it is possible to construct programs for which the heuristics fail and/or the assumptions are violated. In such cases the compiler will revert to simple (non-optimized) code production for the offending expression. In some rare cases, such as the use of unary minus, the compiler may produce incorrect code by interpreting the double-negation axiom in its "permissive" role (see page 23).

One of the assumptions made by DELAY is that the machine has an infinite supply of registers. Thus it assumes that results may be placed in registers. In particular, when a value is used to index into a structure, it is often possible to employ the indexing ability of the hardware to accomplish part of the effective address calculation required. It is convenient and desirable to assume that the index value is in a register. Note that unlike many similar approaches to code generation, DELAY in fact performs no register assignments to real registers; it uses abstract registers referred to (in the BLISS implementation) as "temporary names". It is the responsibility of a later module, TNBIND, to bind these temporary names to physical hardware locations.

The output of DELAY is the tree representation of the program with flags added to indicate sign and location preferences, result types, etc. TNBIND takes this as input, and using the flags added by DELAY, information about the program flow (so that result lifetimes can be determined) and knowledge about the behavior of CODE, binds the temporary names to physical locations. TNBIND knows (in the sense that the knowledge is part of the coding of TNBIND) the cost tradeoffs that will occur when certain results are kept in registers instead of in memory (or on the stack).

The fact that temporary names cannot always be bound to registers means that TNBIND can (and often does) change the estimated cost. In all cases TNBIND attempts to assign registers for the highest-cost computations, and do it in such a way that any other permutation of assignments would incur a higher cost. The determination of these cost figures is a complex process which takes into account the "known" behavior of the CODE module given the characteristics of the PDP-11. To produce a TNBIND module for any other architecture would involve nearly as much effort as constructing the original version. TNBIND cannot, with its current algorithm, produce optimum bindings based upon complex lifetime considerations. The general register-assignment problem[1] is the subject of current research [John74]. However, any general solution must be able to obtain accurate cost data without requiring that the compiler-builder code "by hand" a special machine-dependent routine which performs this calculation.

The output of TNBIND is again the tree representation of the program, with the bindings indicated. CODE takes this as input and produces relocatable[2] machine code. This is presented as a doubly-linked list to FINAL, which performs a set of machine-dependent optimizations based upon the control flow of the machine code and the machine characteristics. This is roughly analogous to the "peephole optimization" [McK65] performed by many compilers and is discussed in more detail in [Wu73]. The output of FINAL is the object program in a

1 As opposed to the general-register assignment problem.

2 In a more abstract sense than "relocatable by the loader". The code has not yet been committed to any particular physical relationship with any other segment of code (see [Wu73], section IV.5).

suitable form for loading[3].

The BLISS/11 compiler structure is used as a model here because it is designed with the goal of maximizing compiler effectiveness by making as much information about the program and machine available <u>at the place where it is needed most</u>. Admittedly, certain heuristics are used where complete algorithms are not known, do not exist, or are prohibitively expensive. The lack of complete algorithms in such places results in incomplete use of knowledge or incomplete generation of knowledge for later use. However, the effectiveness of the compiler is quite high, and the discovery of complete algorithms or better heuristics for generation and use of knowledge does not seem to imply any major restructuring of this decomposition.

One of the major problem areas in the design of a machine-independent compiler is to properly partition out the machine-dependent assumptions which are used in code production. If we are attempting to build a compiler-compiler, we must also be careful to partition out language-dependent concepts.

The classical compiler-compiler method tends to defer the machine-dependent aspects to the code-generation phase of the compiler. It should be pointed out, however, that language-dependent issues (in particular, scope of names) are not usually addressed at a sufficiently abstract level. In the case of scope of names, for example, compiler-compilers either presume static (FORTRAN-like) allocation, or nested (Algol-60-like) allocation. It is typically difficult or impossible to implement complex scopes, such as those of SIMULA-67 [Dah67], OSL [Al71] or ALPHARD [Wu74].

If we intend to produce a compiler-compiler which in turn will produce compilers

---

3 Actually, BLISS/11 outputs symbolic assembler code. The reasons for this decision are irrelevant here.

with a BLISS/11-like structure, it is important to realize that language issues are not easily separable from what have been classically thought of as "machine-dependent" issues. The detection of feasible global optimizations depends heavily upon the ability to change the implied execution order of arbitrary statements. The cases in which the re-ordering is permitted, or is not permitted, depend only upon the formal definition of the language, and should not be considered an integral part of the syntax analyzer.

Specific assumptions about the characteristics of the target machine must also be removed from the implementation. For example, DELAY assumes that the unary complement property of the unary minus operator. As we have mentioned previously, unary minus is in fact _not_ a unary complement operator on any machine using 2's complement arithmetic (as the PDP-11 does). Although the cases in which this assumption is violated are rare (thus justifying its use), there is in fact no way of removing this assumption if the need arises. There is a need to decouple such assumptions from their actual implementation. This is related to the concept of decoupling the _policy_ and _mechanism_ components of a system [Jon73, Wu74]. The collapsing of unary minus operations is a _mechanism_ for performing an optimization; the decision to do this is a _policy_. In the current implementation they are not distinguishable; in future implementations we expect they shall be.

One of the results of our work here is to be able to provide DELAY and TNBIND with specific cost data for each possible code sequence which can be employed to evaluate a node. Since this cost data is derived from a parameterized description of a machine, it now becomes separable from the actual implementation of DELAY and TNBIND; the optimization phase and register allocation phase should then become machine-independent.

Scope of research

The long-term goal of this research is to make it possible to automatically produce a compiler from a language and machine description. In turn, this compiler will produce code at least as good as that produced by the best current optimizing compilers. The immediate goal, i.e., the scope of this thesis, is restricted to the area of machine-dependent code generation.

There are three major aspects of this research. They are: (1) developing a notation which allows the specification of the behavior of machine instructions; (2) developing a set of strategies, using this notation, for code generation in a compiler; and (3) developing a set of methods which may be used in a compiler-compiler to construct machine-independent optimizing compilers.

The need to specify the behavior of a machine instruction is not new. Informal or prose descriptions are the conventional method, and although they are usually readable they are sometimes incomplete, incorrect, or subject to misinterpretation. Some formal notations are biased towards specific machine implementation strategies (such as microprogramming) and in fact are used to derive parts of the implementation directly (such as the microcode). One of the more readable notations is ISP [BN71], which has a form similar to most programming languages[1]. Such notations are designed to present the user with formal, unambiguous descriptions of the behavior of the machine. Our notation is designed with a similar intent, but is biased toward our application, the generation of machine code for a given language[2].

1 For a survey of several such notations, see [Bar73].

2 The notation was designed to be easily manipulated by a compiler or compiler-compiler. The intent was, in fact, that simple encoding techniques would reduce

Common to all compilation methods is the specification of which code sequences realize a given language construct. These are typically specified as code skeletons, called templates, which contain formal parameters. When a template is chosen by the compiler, the actual parameters (memory locations, variable names, etc.) are substituted for these formals. In many code generators, only one or two templates are used for each language construct; however, in any real machine there may be many equivalent code sequences, with varying costs and side effects. The goal of an optimizing compiler is to choose the lowest cost sequence.

One of the problems in constructing an optimizing compiler is deciding which code sequences are semantically equivalent. An error on the conservative side may omit valuable optimizations; an error on the liberal side may produce more efficient code which is incorrect (i.e., nonequivalent in certain cases or for certain values of the data). Given that we have expressed a machine's behavior in our notation we will be able to discover all semantically equivalent code sequences and the conditions under which they are equivalent. We will also be able to attach very specific cost data to each code sequence, thus allowing a compiler to select the cheapest code sequence for known conditions.

The discovery of semantically equivalent code sequences is subject to combinatorial complexity. Many possibilities must be explored before an actual code sequence (or template, in the conventional code-generation sense) is discovered. Thus, we use a preprocessing program to perform the exploration, and its output is a set of

---

the notation to an easily processed bit-level representation inside the machine. Part of this notation has been based on that used in the BLISS/11 compiler [Wu73] where it in fact does have a bit-level representation and thus serves as an existence proof of the feasibility of this conversion.

templates with cost data and input-output relationships attached. The compiler then uses this data to choose the best code sequence for any given language construct. If the compiler is designed in such a way that this choice is based entirely on the data in the templates, with no built-in assumptions about the machine, then it should be possible to substitute a set of templates for another machine and have optimized code produced for that machine. Thus we achieve a substantial degree of machine independence.

Note that the selection of code templates is not the only machine dependent portion of the compiler. The modules for register allocation, machine-dependent code optimizations which are based upon the effects of concatenating codepieces, and interface to the loader (or equivalent) are also involved. Thus we suggest a compiler-compiler solution which attempts to parameterize as many of these functions as possible.

We postulate a compiler-compiler structure such as that shown in Figure 4. Each module in the system requires knowledge of certain properties of the language or machine, and produces a specific piece of the compiler. Note that the output of each module is illustrated as being a portion of the compiler; in fact, it may only be the data necessary for some other translation process to generate the required piece of the compiler. The single-step operation is indicated here for simplicity. The various pieces of knowledge required to construct a compiler are:

Syntax specification

A context-free grammar which specifies the parse rules for the source language.

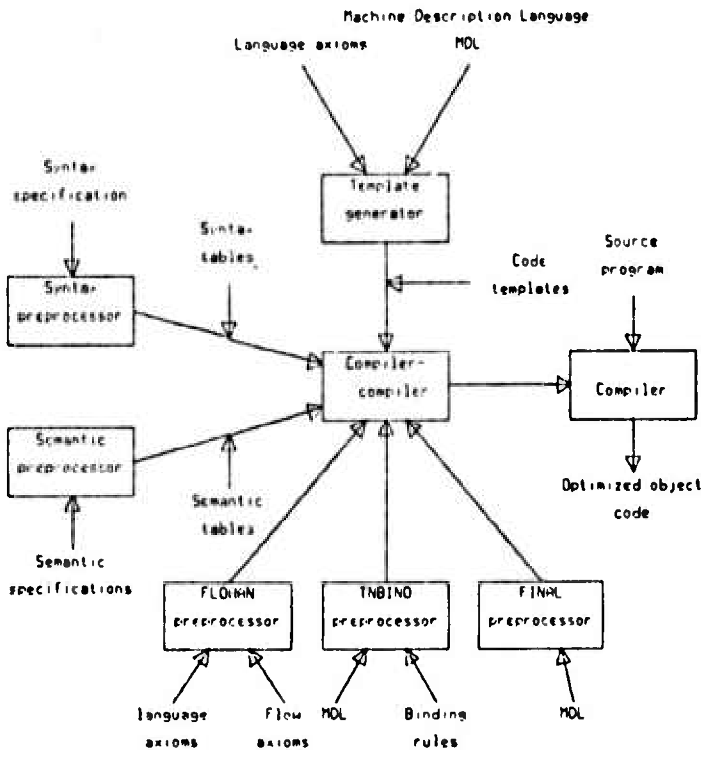

Figure 4: A Compiler-compiler structure

Semantic specification

The specification of bindings, declarations, type consistencies, coercions, etc. necessary to determine that the program is properly constructed, that operators in fact apply to their operands, etc. These semantic specifications <u>do</u> <u>not</u> indicate mappings into machine code.

Language axioms

A specification of the permissible equivalences which may be used in evaluating a program. Generally, these will be axioms about allowable arithmetic transformations, such as '$A+B \equiv B+A$", but can also include certain implementation-dependent specifications, such as "$\forall I , I \leq 32767$" or "$\forall A \mid A \neq -\infty \supset A \equiv -(-A)$".

Flow axioms

These axioms specify allowable alterations the compiler can make in evaluation order, rules for common subexpression detection, and rules for various code motions. The flow axioms could, for example, be expressed in a notation such as that developed by Geschke [Ges72].

Machine Description Language (MDL)

This notation will specify all the relevant information concerning the behavior of the target machine. For the sake of discussion we could consider this the ISP description of the machine [BN71], although in fact we require a more robust and rigorous definition of the machine. Also, the MDL notation must be rich enough to provide the diverse types of information required by DELAY, TNBIND, and FINAL, which take somewhat different views of the world.

Binding rules

The binding rules describe the strategy to be used in determining how to bind real machine resources (such as registers) to the virtual resources allocated by the compiler (temporary names). The relative costs of various storage heirarchies, and the costs of transferring information between them, can be deduced from the (ideal) machine description language.

We consider the development of the general notations for the machine, language axioms, and binding rules to be part of the future research which will follow. Some of the issues involved in the development of machine descriptions suitable for a variety of applications are the subject of current research; for example, see [BS74].

Given that we have a notation, or set of notations, suitable for describing the characteristics of a language and a machine, we must convert the information expressed in these notations into a form suitable for use by a compiler. As illustrated in Figure 4 we assume the existence of programs capable of effecting this transformation. We will concentrate here on the template generator, the key to producing a DELAY module.

The Compiler Model

Given that we have generated a set of templates, we must now devise a compiler structure which uses them effectively. As stated earlier, we have adopted the BLISS/11 compiler structure for the purpose of this thesis. In this context, then, the module we are particularly concerned about here is DELAY. In the current BLISS/11 compiler, DELAY makes a recommendation to TNBIND and CODE, indicating which code sequences will be optimum, assuming that TNBIND can satisfy the assumptions made about register binding. Should these assumptions prove to be infeasible, in the sense that TNBIND cannot satisfy them, then less than optimal code will be produced. It is also the case that the model used by TNBIND to choose the least expensive feasible binding is based upon knowledge about how CODE will actually generate code from the recommendations of DELAY and the bindings of TNBIND. The result is a very large and complex TNBIND, which contains knowledge of CODE distributed widely within itself. Thus any change to the behavior of DELAY, TNBIND or CODE can affect either of the other two modules; the result can be either suboptimal code or incorrect code, depending upon the changes. This interdependence has had a serious impact in two areas of compiler development and maintenance: it has made debugging of the compiler quite difficult at times, and it has made it difficult to predict the impact of any new optimization being added to the compiler.

Our new version of DELAY produces not one recommendation for the entire tree, but a <u>set</u> of recommendations. We could view DELAY as producing a forest of trees, T_1, T_2, ... T_n, for a single input tree. This forest can be ordered so that if C_i is the overall cost of tree T_i, then $i<j \supset C_i \leq C_j$. The only difference between any two trees T_j and T_k is the set of possible bindings which TNBIND will assign.

At each operator node in the tree, DELAY can select one of several code sequences depending upon the state of the operands. Therefore, to choose a code sequence, DELAY (recursively) examines each of the subtrees, and upon return selects all possible code sequences which are feasible for the operands obtained. There is a mechanism, discussed more fully on page 65, which guides the search of the operand subtrees to insure that a maximum number of code sequences will be found.

The behavior of TNBIND is now much simpler. It examines all the alternative trees in the order presented, and for each tree determines whether or not it is feasible. Thus all TNBIND needs to do is choose the first tree for which a feasible binding can be found. Note that in some cases it is impossible to find a feasible binding[1], in which case the program cannot be compiled.

The behavior of CODE is likewise greatly simplified. It only has to perform an endorder treewalk on the chosen tree, collecting the code and substituting the actual bindings for the formal bindings in the templates. The resultant code sequence may now be passed to FINAL for terminal processing.

---

1 Such is the case in BLISS/10, which requires a register for the controlled variable for every INCR/DECR loop. An attempt to nest loops deeper than the number of available registers results in a program which is impossible to compile.

Orthogonal issues

Several issues will not be treated here, because they are either irrelevant or subjects of other research. These may be summarized as follows:

1. Code generation for parallel machines (CDC6600/7600, ILLIAC-IV, STAR).
2. Global program optimization.
3. Register allocation.
4. Formal program equivalence

In constructing programs for parallel machines, particularly the array or stream processors, one is concerned primarily with structuring the algorithms and/or data to take advantage of the architecture. Code generation for such machines involves recognizing explicit or implicit structures in the program which lend themselves to this type of processing, such as the CDC-STAR version of LRLTRAN [Zw75] or the ILLIAC-IV version of FORTRAN, IVTRAN [MM75, PJ75] are being designed to do. Code generation for machines with parallel or pipelined computational units, such as the CDC-6600 or the 360/91 and 360/195 involves organizing the computation so that as many computational units as possible are utilized as efficiently as possible. This sometimes involves knowing the actual data paths used in transferring results internally [IBM69]. These aspects of code generation are complete research areas In their own right, and are the subject of ongoing research elsewhere.

Global program optimization has already been investigated [CS70, Ges72] and, since it is machine independent, is not treated here. Note that the <u>results</u> of global optimization may in fact influence the optimality of local code, and we will consider the results of such optimizations.

Register allocation, or the assignment of particular locations for intermediate results, is an extremely complex problem. It has been worked on since the first FORTRAN compilers, and a discussion of the problem is included in nearly every paper on compiler construction. For certain assumptions, such as commutativity or associativity of operators, particular machine architectures, or absence of common subexpressions, there exist algorithms which minimize code size, register usage, intermediate stores into memory, etc. One such result is given by Sethi and Ullman [SU70], and they cite earlier results.

Register allocation algorithms which minimize code size, intermediate storage into memory, etc. usually work only in the absence of common subexpressions. There are also problems in languages such as BLISS/11 [Wu71, Wu73] where the programmer can make explicit bindings of a name to a register, or where the compiler will implicitly bind a local name to a register. The problem of register allocation is then complicated by the "lifetime" of a result, i.e., the time a result which will be re-used resides in the register. This problem is being investigated by Johnsson [John74], who cites earlier work in the field.

Formal program equivalence, the proof that one program is semantically equivalent to another, is likewise the subject of ongoing research. It is pointed out by Aho, Sethi, and Ullman [ASU70] that some earlier results by other investigators have shown certain equivalence questions to be unsolvable, such as the equivalence of two arithmetic expressions if absolute value and/or trigonometric functions are permissible operators. The problems of formal proofs of program equivalence are so complex that there are not yet any "practical" results. The classes of machines, problems, and results are still too restricted to have applicability in compiler construction for real languages on real machines.

While all of these topics bear on the topic of this thesis, they are not treated here; each represents a major research area in its own right, which can be explored independent of this research. Although we will use the results of research in these areas in the construction of our ultimate compiler-compiler system, the scope of this thesis is such that most of these issues are not immediately relevant.

Chapter II

Methods

Introduction

In this chapter we shall develop a notation which can be used to express the behavior of machine instructions. By using this notation, we will illustrate a set of methods to be used in discovering code sequences for compiling a given language construct. In the next chapter we shall show how this notation has been used to implement a prototype system for discovering code templates.

Introduction to attributes

We shall characterize the behavior of machine instructions in terms of their input-output relationships. Each machine instruction has a set of conditions which must be true before it can be executed, and it produces a set of conditions as the result of its execution; these are often referred to as "preconditions" and "postconditions". We wish to describe these preconditions and postconditions in a form that can be manipulated by another program in order to find a sequence of instructions with the desired overall behavior.

The techniques used in this thesis are standard methods of artificial intelligence. Although simple, they have demonstrated the validity of the proposed solution. Investigation of methods for general problem solving lies outside the scope of this thesis.

The method used here follows the paradigm of means-end analysis given by

Ernst and Newell for GPS [EN69]. The basic problem faced by such a program is to take a state description which represents an initial state, and a state description which represents a "solution" state, and transform the initial state to the solution state. The operations required to effect this transformation then represent a solution of the problem.

The technique is applied recursively. The program, in a given state, examines the current state of objects and compares this state to the desired state ("goal state"). If it discovers that these two are not "equivalent" then it searches for an operator which will reduce the difference between the current state and the desired (goal) state. A set of such operators may exist; some criterion is applied to decide which, if any, of these operators are to be applied to the current state to achieve to goal state. Note that although an operator may reduce the difference between the current state and the goal state it may not directly achieve the goal state; furthermore, there may be no other operators which can act upon the reduced state to reduce the differences still further. In this case it is necessary to "back up" and attempt another operator (if available) to reduce the difference between the current state and the goal state. This process will continue until there is no longer any difference between the current state and the goal state, or until no operators remain to be tried. In the former case the succession of operators represents the complete reduction operator from the current state to the goal state, and may now be stored away as a single operator, if desired; in the latter case, failure is reported, and the search will return to a higher level (previous state) if one exists, otherwise there is no solution. The initial state description and the "solution" state description are taken as the initial "current" and "goal" states, respectively.

We do not implement GPS to solve this problem. Indeed, to use the "pure" GPS paradigm requires that the program discover a great deal about what it is trying to solve. Instead, our system contains a great deal of specific knowledge about its task domain and how to explore it. In addition, information is stored in rigidly defined structures, and the knowledge of these structures is an integral part of the program.

A significant deviation from the GPS approach is that we cannot accept _any_ solution, but must discover the _best_ solution. The optimum solution may not be the first solution found; in particular the optimum local solution may not be the optimum global solution, so we must discover _all_ solutions. It is not until an actual program is compiled that sufficient information is available to choose any one particular solution over the alternatives.

The objects we operate on are parse trees[1] which have been processed through some global optimization phase of a compiler. There is no loss of generality in assuming that the tree is structured as in the BLISS compiler [Ges72, Wu73]; indeed this structuring is more general than that encountered in most other compilers. At each node of the tree we have the typical representation: an operator for non-leaf nodes, with _n_ descendants for each of the _n_ operands of the operator; the name (or a symbol table pointer, or some other equivalent datum) of the symbol for leaf nodes. In addition to this conventional information, we include a set of property-value pairs at every node which describe relevant properties of the node. Typical examples might be the location of the value computed at the node (memory or register) and whether or not the value is required at a later time, or can be destroyed once it is used (common subexpression).

[1] Or directed acyclic graphs (dags). Although the current implementation in fact only generates trees this is not an inherent restriction in the system.

In order to distinguish the operators which manipulate objects during the search from other types of operators, we will make the following distinctions: operators which manipulate objects during the search are called _transformation operators_ or _T-operators_; operators of a source language are called _language operators_ or _L-operators_; and operators in the machine we are considering are called _machine operators_ or _M-operators_.

T-operators on objects are of two kinds: those which compute the result of applying the L-operator to its subnodes, and thus represent an evaluation of the tree, and those which transform a subnode into a form which can be operated upon. This latter form of T-operator is similar to the _transfer function_ of Hopgood [Hop69, pp 78f]. Hopgood specifies transfer functions as properties of unary operators (not necessarily unary complement operators[1]; such unary operators as FIX, FLOAT, ABS, and -ABS are included). Transfer functions are applied to the tree representation to delay the actual evaluation of the unary operators; the result is that certain unary operators become absorbed (in much the same manner as unary complement operators) and need never be explicitly applied.

When we examine a machine architecture, we find that many M-operators are defined only over a restricted domain. In the mathematical sense, arithmetic operators are defined only for results which can be represented within the finite precision of a machine word. However, this is only a simple example of why M-operators are

---

1 A _unary complement_ operator is an operator which when applied twice to the same operand produces no net effect. In common algebra, such as the field of real numbers, unary minus is a unary complement operator, i.e., $-(-A)\equiv A$; in Boolean algebra it is the "not" operator: $\sim(\sim A)\equiv A$; over the field of real numbers we also have the multiplicative inverse: $(1/(1/A))\equiv A$. Specific use of the unary complement operators will be discussed later. See also Frailey [Fra70].

"partial" functions. In a conventional general register architecture, an M-operator such as "ADD" is defined only if one of its operands is in a register and the other in memory. If both operands are in registers, then there might be another M-operator, "ADR", which is defined for this condition. One of the functions of a compiler is therefore to map the operands of an M-operator into the location range over which the M-operator is defined.

We associate with each L-operator a set of M-operator sequences which effect the computation desired; for example, the machine-language instruction (M-operator) "ADD" is associated with the source-language operator (L-operator) "+"[1]. We then explicitly state the domain over which the M-operator is defined. As we process the tree, if we find that the subnodes of a given node fall within the domain of the M-operator sequence associated with the L-operator at that node, then we can state that the M-operator sequence represents the "compilation" of that L-operator. The problem becomes more complex when the subnodes do not fall within the domain of any M-operator sequence associated with the node. This, in fact, is the typical case encountered during compilation.

In the case where one or more subnodes of a given node do not fall within the domain of (any) M-operator sequence which evaluates the node, we search for some way of transforming the offending subnodes into a usable form. To accomplish this we compute the difference between what we have (current state) and what we need (goal state). We then search for some "difference reducing operator" which effects the transformation. For example, if we have an ADD instruction which requires one of its

1 The method for determining this association constitutes an entire research problem in its own right, and discussion of it will be deferred until a later section.

operands in a register, and both of its operands are in memory, then one of the possible difference reducing operators is that one which transforms an operand in memory to an operand in a register, e.g., a LOAD instruction.

One of the desired goals of a compiler is to produce the minimum cost[1] code sequence (M-operator sequence) which evaluates a given tree. It is therefore necessary to explore all possibilities which represent evaluations and eliminate those which exceed the least-cost solution and are semantically equivalent to it. This semantic equivalence is also related to the effect of a machine instruction (M-operator) on the global program state in the context in which the M-operator sequence is executed. It is therefore necessary to express the global program state conditions under which an M-operator sequence may be applied, and the resultant transformation in this state.

Note that achieving this goal results in a deviation from the normal GPS goal-search technique. In GPS, any operation which reduced the difference between the initial state and the goal state, such that the goal state could eventually be reached, was satisfactory. Here, we have the additional constraint of requiring a minimum-cost transformation. The determination of minimum cost is complicated by the fact that it depends upon a context more global than that of any single node. The minimum code sequence for a subtree may not result in optimal evaluation of its parent tree; what appears to be suboptimal code for a subtree may be, overall, more efficient. Note particularly in the context of the BLISS/11 compiler structure that absolute costs

[1] Note that "cost" is treated in most of this work as an abstract concept. It could be memory cycles, code size, register requirements, or something even more complex; only the concept of cost is involved, not a specific set of parameters to be measured

cannot be determined until TNBIND has assigned locations to temporary results, and that this allocation depends upon accurate knowledge of the cost that each alternative allocation will incur. We cannot, therefore, accept either the first sequence which satisfies the expression, nor the locally best sequence; it is necessary to obtain all sequences (modulo semantic equivalence). The general solution to the determination of semantic equivalence of code sequences is another complete research area and outside the scope of this thesis; however, it has been shown for some cases cited in [ASU70] that the problem is unsolvable. The current implementation assumes all sequences are unique.

Global side effects introduce an additional dimension of freedom. All M-operators have side effects, but not all of these are relevant. For example, an ADD instruction sets the carry and overflow bits, but if there is no test of these bits in the program the side effect is not relevant. The relevance of such side effects depends upon context and therefore changes from site to site within the program.

It should now be obvious why we are approaching this from the viewpoint of a compiler-compiler. The entire process just described is much too slow to be included in a compiler. Compilers must be reasonably efficient, and not much more complex than a simple finite-state automaton when processing each node during code generation. The research described here represents only one small portion of an actual compiler-compiler. The task of the system described is to derive the templates which would be used in some fairly conventional code generator. However, since it does this derivation by exhaustive search, it is more likely to find all the obscure cases and unsuspected equivalences than most hand-designed sets of templates.

It is pointed out by Wulf (who is certainly not the first to discover it!) that the generation of code templates (specifically, the case analysis required) for a real compiler "represents a substantial amount of intellectual effort, has been modified many times as new cases were uncovered, and still has no guarantee of being exhaustive" [Wu73, p.81]. Extensive case analysis was required, to guarantee both efficient and <u>correct</u> code. In particular, when the compiler generated incorrect code it required extensive effort to locate where an incorrect optimization was first chosen. We hope to substantially reduce this effort by not choosing incorrect ootimizations!

Attributes

Attributes (informal)

Motivation for using attributes

In the next section we formally define attributes. The intent of this section is to provide an informal motivation for their need and use.

It was previously mentioned that M-operators are partial functions over both the values they operate on and the locations in which the values may be stored. Compilers do not generally concern themselves with the specific values on which they operate, but leave this to the discretion of the run-time support system. However, optimizing compilers must be concerned with the types of locations in which values may be stored, with certain abstract properties of these locations and values, and with the sets of M-operators are applicable to these situations.

We therefore assume that all we need to examine is a tree on which all global optimizations have been performed. We need to produce a set of directives to the compiler which determine what code to emit for a given construct, given the properties of the M-operators and the state that the object program will be in when they are executed. Several existing systems provide notation for this, including TREEMETA [CLE69] and the systems described by Miller [Mil71] and Donegan [Don73]. The use of such systems poses much the same problem for the user as most of their predecessors (although, in the case of the last two, not to the same degree), i.e., complex case analysis is required to use them correctly and effectively. We intend to perform this case analysis exhaustively; this is based on the premise that two solid weeks of computer time is cheaper than six man-years of human time, especially since the task, when performed manually, involves a substantial amount of debugging time, which is neither creative nor interesting. It will be seen later that the actual case analysis only involves a few thousand cases for each L-operator.

It should be noted at this point that we seem to be describing a system for producing input to a compiler-compiler, or in fact a compiler-compiler-compiler. Beyond a certain level the description of such distinctions becomes quite complex; for a technique used for describing such interactions see the paper by Early and Sturgis [ES70]. Until we find it necessary to talk in detail about the output of the system we are building, it is convenient to assume that its output is the code generator of the ultimate compiler, instead of the data used to create that code generator.

In order to analyze the trees, it is necessary to characterize the problem in a way which makes it amenable to purely mechanical analysis. We therefore examine the requirements of the M-operators first. A small set of properties characterizes an M-operator. Typical properties include the type of machine location (e.g., register or memory) where its operands can or must reside, the sign of the result relative to its expected sign (i.e. the treatment of unary minus), whether or not the M-operator destroys its operands (M-operators of single-address and two-address or general register machine architectures), and whether or not it affects the program counter (SKIP and branch instructions) or condition codes (if such exist).

We shall refer to such properties as "attributes". The prototype system depends heavily upon this concept. However, it is important to note here that the system understands only the concept of attributes. There are no specific attributes built into the system. Thus "fundamental" concepts such as registers can be totally ignored, and either a two-address memory-to-memory machine, a stack machine, or a three-address machine could be handled with equal facility.

The result of treating such concepts as registers, program counters, condition codes, etc. as abstract properties of the machine is that there is no commitment to any particular machine architecture. The existence of zero, one, or more general registers, a single program counter, and similar assumptions are not built into the code generator. This is a significant step toward a machine-independent compiler structure.

The selection of a set of attributes is thus a function of the choice of target machine, not an inherent property of the language or compiler, i.e., for a machine M chosen from the set $\mathfrak{M}$ of all possible machines and for $\mathcal{A}$ the universe of all possible attributes, $\forall m \mid m \in \mathfrak{M}, \exists A \mid A \subseteq \mathcal{A}$ which defines the relevant attributes for a given machine m; it is certainly not the case that for any set $\mathcal{A}$ that $\mathcal{A}$ will be applicable to all machines.

In the next section we define a formal relationship, designated $\leq$, between sets of attributes. Informally, the relationship may be characterized as a "more general" relationship; if $A \leq B$, then B is considered "more general" than A. This relationship is used to search for code sequences, compare current states to goal states, and related applications.

If we consider a general register architecture for the sake of illustration, we might consider the question of where to leave a temporary result. The process of binding a temporary name to a physical location is the function of TNBIND. The DELAY module may have indicated that the name should be bound to a register, or that it may be bound to either a register or a stack location; we could indicate this by stating "$loc_{node} \in \{register\}$" or "$loc_{node} \in \{register, stack\}$". The latter constraint is more general, and we could indicate this by writing $\{register\} \leq \{register, stack\}$. This is only one example of the use of the $\leq$ relationship; it is more complex when the properties represent disjoint attributes.

With this as background, we now give a formal description of these attributes.

Attributes (formal)

The property-value pairs which we associate with each node are called attributes. We will now give a formal definition of these attributes.

Let $\mathcal{N}=\{ N_1, N_2, \dots \}$ where each symbol N_i is referred to as an attribute name.

Let $\mathcal{V}_i=\{ V_{i1}, V_{i2}, \dots \}$ where $i \in \mathcal{N}$. The symbols V_{ij} are referred to as attribute values.

Let $\mathcal{AV}=\{ <n,V> \mid n \in \mathcal{N}, V \subseteq \mathcal{V}_n\}$. Each pair $<n,V>$ is a particular attribute with a set of values and is referred to as an attribute-value pair or A-V pair.

We define a partial ordering $\leq$ over subsets of each $\mathcal{V}_i$ by the common subset relationship $\subseteq$[1], i.e.,

for $V_1, V_2 \subseteq \mathcal{V}_i$, $V_1 \leq V_2$ iff $V_1 \subseteq V_2$.

Note that $\leq$ is not defined across value sets with different indices, e.g., for some $V_1 \subseteq \mathcal{V}_i$ and $V_2 \subseteq \mathcal{V}_j$, $i \neq j$ implies $V_1 \leq V_2$ is undefined[2].

We define the symbol "=" such that for any sets X, Y: $X = Y$ iff $X \leq Y \wedge Y \leq X$.

We define a partial ordering $\leq$ over members of $\mathcal{AV}$ as given below. Note that

1 We use this definition (rather than the subset operator $\subseteq$ primarily for an implementation reason: the partial-order predicate is implemented in such a way that it can accept any two operands of the same type (value sets, attribute-value pairs, or attribute sets, and will always return the correct result. It is thus meaningful to use a single operator in the external representation as well.

2 Note that the tokens used to designate values are always unique. If we define $\mathcal{V}_1=\{A, B, C\}$ and $\mathcal{V}_2=\{A, B, C\}$, it is impossible to determine the meaning of the relationship $\{A, B\} \leq \{A, B, C\}$. Internally, these value designators would be unique, i.e., $\mathcal{V}_1$ would really be defined as $\{A_1, B_1, C_1\}$ and $\mathcal{V}_2$ as $\{A_2, B_2, C_2\}$. Thus, the only predicates that could be posed would be of the form $\{A_1, B_1\} \leq \{A_1, B_1, C_1\}$, which is meaningful, or $\{A_1, B_1\} \leq \{A_2, B_2, C_2\}$, which is undefined. This rather sticky problem can be avoided by choosing unique print names for each of the values to be considered, or using typed sets.

the same symbol will be used for all partial orderings; it will always be clear from the context what is being compared.

For $Q_1, Q_2 \in \mathcal{AV}$, and letting $Q_1=<n_1,V_1>$ and $Q_2=<n_2,V_2>$, we define $Q_1 \leq Q_2$ iff $n_1 = n_2 \wedge V_1 \leq V_2$.

Let $\mathcal{AS}$ be any set defined as

$$\mathcal{AS}=\{ <n,V> \mid <n,V> \in \mathcal{AV}\}$$

such that

$$\forall <n_1,V_1>,<n_2,V_2> \in \mathcal{AS},\ n_1=n_2 \rightarrow V_1=V_2$$

Note that $\mathcal{AS}$ is a function in the mathematical sense; for each attribute name in the pairs of the set there is one and only one possible set of values associated with that name. We will refer to sets of this nature as <u>attribute</u> <u>sets</u>.

We wish to extend the partial ordering relationship to attribute sets. However, we must first impose the requirement that both of the attribute sets being compared contain exactly one occurrence of an attribute-value pair for each name in the set $\mathcal{N}$. In many cases we are interested in only a subset of the attribute-value pairs, and the remainder are "don't care" conditions. We can accomplish this by extending the sets with pairs of the form $<n_i, \mathcal{V}_i>$ for all n_i not in the original set. We define the closure, $\mathcal{C}(\mathcal{AS})$, of any attribute set $\mathcal{AS}$ formally as

$$\mathcal{C}(\mathcal{AS})= \mathcal{AS} \cup \{ <n, V_n> \mid \sim (<n, X> \in \mathcal{AS}) \text{ where } X \subseteq \mathcal{V}_n\}$$

We can now define a partial ordering $\leq$ over attribute sets as follows:

$$\mathcal{AS}_1 \leq \mathcal{AS}_2 \text{ iff } <n,V_1> \in \mathcal{AS}_1 \rightarrow <n,V_2> \in \mathcal{C}(\mathcal{AS}_2) \wedge V_1 \leq V_2.$$

As a convention in the implementation, we make attribute names distinguishable symbols. Attribute names are denoted by names beginning with the symbol "\$". For convenience, we allow a singleton set of attribute values to be represented as a single name, e.g. MEM ≡ {MEM}.

Some examples illustrating the ≤ relation are given below. Recall that the symbol "≤" may be used to apply to sets of values, attribute-value pairs, or attribute sets. We allow subscripts of sets to be symbolic names, not just numbers; values are designated by symbolic names. For this illustration we define

$\mathcal{N}$ = {\$LOC \$SIGN}

$\mathcal{V}_{loc}$ = {REG MEM CC}

$\mathcal{V}_{sign}$ = {+ -}

Over some typical value sets, the relationships are:

{REG} ≤ {REG MEM}
{REG} ≤ {REG}
{REG} ≤ $\mathcal{V}_{loc}$
{REG MEM} ≰ {REG}
{REG MEM} ≤ {REG MEM CC}

Using the same sets as above, over attribute-value pairs,

<\$LOC {REG}> ≤ <\$LOC {REG MEM}>
<\$LOC {REG}> ≤ <\$LOC {REG}>
<\$LOC {REG MEM}> ≰ <\$LOC {REG}>

Note that a test such as

<\$LOC {REG MEM}> ≤ <\$SIGN {+ -}>

is undefined because the attribute names differ.

We can express the closure set $\mathcal{C}$ as

$\mathcal{C}(\phi)$ = {<\$LOC {REG MEM CC}> <\$SIGN {+ -}>}

(where ϕ is the empty set) and thus express relationships over attribute sets as

{<\$LOC MEM>} ≤ {<\$LOC {REG MEM}>}
{<\$LOC MEM>} ≤ {<\$LOC MEM>}
{<\$LOC MEM> <\$SIGN +>} ≤ {<\$LOC MEM>}
{<\$LOC MEM>} = {<\$LOC ϕ>} = {}
{<\$LOC MEM>} ≤ {<\$LOC MEM> <\$SIGN ϕ>}
{<\$LOC MEM> <\$SIGN +>} ≤ {<\$LOC ϕ> <\$SIGN {+ -}>}
{<\$LOC MEM>} ≰ {<\$LOC REG>}
{<\$LOC ϕ>} ≤ {<\$LOC MEM>}
{<\$LOC ϕ> <\$SIGN +>} ≤ {<\$LOC REG> <\$SIGN ϕ>}

<u>Notational abbreviation</u>

Part of the actual notation used in the implementation will be explained here because it will be used in many subsequent examples. The choice of representation was influenced strongly by the use of LISP as the implementation language, and at the implementation level it reflects many conventions of the LISP language, which are not of interest here. The "formal" notation, with its many levels and different types of brackets, is rather cumbersome to write and read. Since the names can be delimited by recognizing the "$" symbol, we typically omit the pairing brackets, "<>". We also replace the set symbols for the attribute set by square brackets, and the set symbols for the values by parentheses. The resultant notation is not only more readable, but it is closer to that actually used in the implementation. Thus we represent

```
{<$LOC {MEM REG}> <$SIGN {+ -}>}
```

as

```
[ $LOC (MEM REG) $SIGN (+ -)]
```

The empty set is denoted by the LISP atom NIL. The universal set can be obtained from the closure set of NIL.

Operations on attributes

In this section, we will define one representation of attribute transformations. This representation is used in the prototype system, and is satisfactory for the GPS-like model used. The choice of some other heuristic search technique would influence the representation, as well as the implementation system chosen. The representation used if PLANNER [Hew72] or CONNIVER [SM72] might be considerably different. This representation is presented here because it illustrates a concrete use of the concepts of attributes introduced in the previous chapter, and it also demonstrates some of the factors which must be taken into account in any implementation.

We will also introduce the concept of a "preferred-attribute set", a technique used to guide the search for a machine operator. This concept is extremely powerful when used by the BLISS/11 compiler in DELAY. Our proposed compiler will also use it, and its generalization and formalization at this level is necessary for understanding the behavior of the compiler. It also means that a fairly powerful heuristic tool is available to whatever system actually searches a machine description for desired code sequences.

Attribute transformations

The attribute sets describe certain properties of the nodes of a tree. The name associated with any leaf of a tree is merely a token, and has no significance to the search algorithms. These concern themselves only with the attributes of the leaves. For any non-leaf node, both the L-operator and the attributes at that node are used by the search algorithms.

In order to avoid the problem of whether an address or the contents of an address are required for the evaluation, the tree representation of the program must contain an explicit dereferencing operator to obtain the contents of a node. This closely follows the semantics of several languages, including BLISS [Wu70]. There is no loss of generality in this assumption since it is always possible for the syntax analyzer to place this dereferencing operator in the tree as it is constructed. For notational convenience, since BLISS will be used for most of our examples, we will use the BLISS dereferencing operator, ".".

<u>The T-operators</u>

A T-operator is defined as a 7-tuple,

$$< \mathfrak{K}, \mathfrak{P}, \mathfrak{R}, \mathfrak{C}_{pre}, \mathfrak{C}_{post}, \mathfrak{M}, \mathfrak{S} >$$

where we define the elements to be from the following sets:

$\mathfrak{K}$	The set of retrieval keys for the operator.
$\mathfrak{P}$	The patterns (parse trees) to be matched against the current state. If the pattern match is successful, then the operator may be applied to the parse tree.
$\mathfrak{R}$	The "result" pattern. If the operator is applied to the parse tree this describes the resulting node of the tree.
$\mathfrak{C}_{pre}$	This is a set of predicates which must be true (simultaneously) of the program state before the target compiler will be able to apply this T-operator. Although this is of no concern of the compiler-compiler, it is necessary for the target compiler to be able to recognize what global conditions influence the validity of a T-operator.
$\mathfrak{C}_{post}$	If the operator is applied, this is a set of predicates which describe the

resultant transformations in the global program state which the compiler must be aware of. These are described as a set of assertions about what is now true about the global program state.

$\mathfrak{M}$ The set of machine instructions (possibly empty) which are required to evaluate the node in the object code representation of the program.

$\$$ The "cost function" data used to determine minimum-cost code sequences.

Retrieval keys are of two varieties in our implementation. First, we define the set of language operators Ω_{lang}:

$$\Omega_{lang} = \{ x \mid x \text{ is an L-operator}\}.$$

This will allow us to retrieve T-operators for evaluating a given node. We will also define a set $\mathcal{V}^r$ of "retrieval values", a generalization of attribute sets. The precise definition in our implementation will be given later; at this point it is sufficient to say that the purpose of retrieval values is to make it possible to locate code sequences which effect transformations of a nodes attribute set --for example, the transformation of a value in memory into a value in a register. Therefore, we will define

$$\mathcal{K} = \Omega_{lang} \cup \mathcal{V}^r.$$

If, for some $p \in \mathcal{K}$, if $p \in \Omega_{lang}$ then the T-operator represents a code sequence which evaluates the source language operator (L-operator), p, assuming the necessary conditions are met. If $k \in \mathcal{K}$ and $k \in \mathcal{V}^r$ then the T-operator represents a code sequence which transforms the node being examined, again if the necessary conditions are met.

In order to evaluate a parse tree (which we will refer to as the source tree), we take the L-operator of the root node, call it q, and use it as a retrieval key to select a set of potentially applicable T-operators. For each T-operator we then compare the source tree to the pattern tree contained in the T-operator (i.e., a member of $\mathcal{P}$). The conditions of applicability are met if (1) the trees have the same shape, (2) the operators at each node are identical, and (3) the attribute sets of the source tree are $\leq$-related to the attribute sets of the pattern tree.

We compare tree shapes by performing a preorder treewalk in parallel on both trees, and compare attributes by performing an endorder treewalk on both trees. This requires only a single traversal of the tree [Kn68, p 316ff][1]. During the recursive descent into the comparison of the source tree and the pattern tree we do not compare attribute sets of the nodes; this is performed during the return upwards in the trees. It is meaningless to perform the comparisons until the subtrees of a node have been evaluated, since their results can in fact alter the attributes present at the node. The attribute sets are compared as part of the "evaluate root node" operation of the endorder portion of the treewalk. If the attribute sets do not match, i.e., are not $\leq$-related, a differencing operator is applied, and the resulting difference indicates what attributes must be transformed in order to make the operator applicable.

Once we have met the above conditions, the compiler-compiler can apply the T-operator. However, this only produces a template to be used by the actual compiler. During compilation, several other conditions may determine whether or not the template can be used. These are described in the precondition set, σ_{pre}. These

1 A preorder treewalk is "root, left, right"; an endorder treewalk is "left, right, root". If we use the sequence "root, left, right, root' " we can perform the preorder functions during the "root, left, right" and the endorder functions during "root' ".

include conditions both of the internal state of the compiler and the defined external state of the program. For example, there is a T-operator which transforms a value in memory (.M) to a value in a register (.R) and incurs no cost, providing the assertion "VALUE(.R) = VALUE(.M)" is true, i.e., the value already exists in the register. Similarly, the optimization $.A \equiv -(-.A)$ can be applied if the assertion "VALUE(.A)$\neq -\infty$" is true for all values of .A (where $-\infty$ is the largest negative number which can be represented in the machine). Note that such an assertion would not be necessary on a machine with a representation other than two's complement; in this case a different set of assertions would be required, for example, to handle the case of negative zero in a signed-magnitude representation.

When the compiler-compiler applies the T-operator, the resulting effect on the tree is described by giving a new tree. The entire set of such results is the result set, $\mathfrak{R}$. The result pattern can contain "active" parts, which in the implementation are LISP λ-expressions. A typical such result would be that which changes the sign of a node: it would have a codepiece associated with the $SIGN attribute which we could define as

$$\text{\$SIGN: } `+' \rightarrow `-';\ `-' \rightarrow `+'\ ^{1}.$$

If we call the set of all such codepieces $\mathfrak{C}$, then the set of attribute values for a result pattern for attribute i, $\mathcal{V}^r{}_i = \mathcal{V}_i \cup \mathfrak{C}_i$. I.e., for all attribute-value pairs $<A, V_i>$, we allow V_i to belong to the set $\mathcal{V}_i \cup \mathfrak{C}_i$.

1 Such active parts are implemented as LISP λ-expressions. For notational convenience this codepiece may be named; thus two more of the attributes of $SIGN are "OPPOSITE" and "SAME". For those who are interested in such things, these are stored on the property lists of the identifiers OPPOSITE and SAME under the indicator ATTRIBUTE-LAMBDA.

If the compiler finds that the specified preconditions are met, it applies the T-operator, producing a code template and determining the effects upon the program state. The set of code templates, as mentioned before, is $\mathfrak{M}$. The effects on the program state are described in the set of output conditions, $\mathfrak{C}_{post}$. For example, a T-operator which transforms a result in memory to a result in a register has the output condition the assertion that VALUE(.R) = VALUE(.M).

The cost of the code emitted, a member of the set $\mathfrak{S}$, can be thought of as a function which is evaluated and yields an integer result. There are many ways of implementing this, such as building another active codepiece into the tuple, or simply building a vector of integers and incorporating the cost function into the compiler-compiler and/or the result compiler. These methods may be considered isomorphic, along with many others, and therefore we will not define the actual appearance of a cost function until we get to specific examples.

The reason the cost function is undefined is to allow the compiler builder to choose what constitutes "cost". It is therefore possible to build a compiler which optimizes only code size, or register usage, or memory references; it is equally possible to build a compiler which optimizes along one of these dimensions which can be selected by the coder of a source program (ideally, for any part of the program). Furthermore, there is no reason to restrict the optimization to a single dimension, or even to be statically defined. However, if the cost depends upon context that cannot be determined until a program is compiled, then there are fewer opportunities to detect semantically equivalent code sequences during template generation, and thus the compiler may have to search many more (possibly redundant) templates to locate valid code sequences.

Preferred-attribute sets

One of the powerful techniques used in the BLISS/11 compiler is the use of "preferred-attribute sets" (PAS). A PAS allows an ancestor node in the tree to specify to its descendant nodes which attributes it would prefer the descendant to return for the evaluation of its result. Thus, if it is desirable for a node to produce a value whose sign is the opposite of that expected, the compiler will perform a case analysis based on considerations such as its operator type and the amount of code required to change the sign of its result, and determine which signs would be best for its descendants to produce. It then passes this information to the descendant nodes, who will attempt to comply with the request. Note, however, that if they fail to comply, they return whatever they have to the parent node, who then has the responsibility of deciding what to do with the results. It may well be that it cannot satisfy the request made by <u>its</u> parent, and the whole process is repeated higher in the tree.

The power of the PAS lies in its ability to pass global context information down into the tree. Without this global information, it would be necessary for each node to assume that the least general result must be produced; e.g., for a + node the compiler would have to assume that a full word bit representation with proper sign is necessary. In fact, the parent node might be willing to accept any sign, or it might require only an address; and in both cases the compiler would have more freedom to choose a code sequence and thus allow for a less expensive code sequence. The PAS also can restrict the type of result by insisting that it be a full-word bit representation with proper sign, in which case it reduces the alternatives which must be examined and thus prunes the case analysis.

BLISS/11 as implemented uses only a very small number of attributes in its PAS. These include the sign bit, the name of a register which is to hold the result (this is known, in the BLISS/11 terminology, as "targeting"), the context in which the result may be used (e.g., as a operand, as an address, etc.) and the "type" of the result: real (a bit representation of a value), flow (a change in the program counter), both, or neither (equivalent to the Algol-68 concept of "voiding" [LvdM73]). Our choice of attributes is similar, but there are several extensions and some special cases are subsumed into more general cases.

The creation of a PAS is highly machine-dependent. It requires knowledge of the instruction set and its symmetries (or asymmetries), the addressing modes of the hardware, and the relative costs of computationally equivalent sequences which have the same external result but different internal results, such as leaving a result in a register, in memory, or on the stack (the result is the same; the cost of accessing it may be different).

The compiler we envision will have a structure similar to that of the BLISS/11 compiler, but the implementation of DELAY, TNBIND, and CODE will be substantially different. All machine-dependent knowledge will be separated out of the actual code into a set of tables. One of the central tables is the code template table.

We will represent code templates as T-operators[1]. The "pattern" component indicates which attributes are required of the subnodes, and the "result" component indicates what attributes the node will return, given the indicated attributes from its subnodes. The "cost" component is the cost of the code under these conditions.

---

1 For those who are concerned with efficiency issues, please note that most, if not all, of the information of a T-operator may be packed in some suitable form for efficient manipulation. The "retrieval key", for example, may be used simply to thread T-operators of keys onto a list. The order of this threading could be chosen in some way to minimize search time.

When we perform the treewalk in DELAY we pass control to a node-specific routine each time we descend in the tree. One of the parameters to this routine is a PAS. A PAS is an ordered list of attribute-cost pairs, sorted in ascending order by cost. If we let α be an attribute set, and c be a cost value, a PAS appears in the form:

$$< (\alpha_1, c_1), (\alpha_2, c_2), \ldots ,(\alpha_n, c_n) >$$

where c_i is less than or equal to c_{i+1}. Each c_i represents the cost the parent node will incur if the subnode returns an attribute set α_i.

In order to explain how this is accomplished, it is necessary to introduce some notation at this point. We define each descendant of a node to have a named path by which it can be reached, e.g., LO: will reach the left operand of a binary node and RO: will reach its right operand. It is useful for expository reasons to be able to apply these path names to any structure related to the tree. Thus, a binary operator is represented as a T-operator, call it T_0. The "pattern" part of T_0 may be accessed by some pathname, such as PAT, applied to T_0 (e.g., $\mathfrak{P}^*$: T_0), and the desired attributes of the left subnode of the T-operator may then be accessed as LO: ($\mathfrak{P}^*$: T_0). As a notational convenience, we will assume the composition rule A_1: (A_2: ...(A_n: X)...) may also be written without parentheses as A_1: A_2: ... A_n: X, for any pathnames A_i and object X.

We generate a PAS in the following manner: given the pathname of a subnode, SN:, we wish to generate the PAS for SN:. We first form a PAS (which we will designate the "uncollapsed PAS") by forming the ordered set $\mathfrak{UP}$ from the set of T-operators for the node T_0, $\mathfrak{T}$:

$$\mathfrak{UP} = < <\text{SN: } \mathfrak{P}^*\text{: } t_i, \text{COST: } t_i> \mid t_i \in \mathfrak{T} \wedge \text{COST: } t_i \leq \text{COST: } t_{i+1} >$$

That Is, the uncollapsed PAS is a set ordered by cost and consisting of attribute-cost pairs, where the attribute is selected from the pattern component of all the templates, for the pathname to which we are about to pass control. We then partition the uncollapsed PAS into equivalence classes based upon cost, where each equivalence class E_i is defined as

$$E_i = \{ \alpha_j \mid \langle\alpha_j, \mathrm{COST}_j\rangle \in \mathfrak{U}\mathcal{P} \wedge \mathrm{COST}_j = i \}$$

Thus E_0 is the equivalence class of all zero-cost code sequences, E_1 the class of all sequences of cost 1, etc.

Within each equivalence class we collapse attributes by forming the union of all attribute sets which differ in no more than one attribute-value pair. The result is a new equivalence class of the same cost. The actual method of collapsing is simpler to represent as an algorithm[1] than as a formal description:

[1] The algorithm is coded in a notationally convenient hybrid of BLISS, LEAP [FR69] and SAIL [VL73]. The only construct which may require some explanation is the <u>leave</u> statement, which causes control to exit the block named by the <u>leave</u>. In the algorithm given, "<u>leave</u> LOOP" causes control to pass to the first statement following the loop, thus providing a premature termination of the loop.

```
procedure collapse.attributes( $E_i$ );
    begin
        set RESULT, E2, A1, A2;
        RESULT $\leftarrow$ E2 $\leftarrow \phi$;
        while $E_i \neq \phi$ do
            begin
                A1 $\leftarrow$ a1 | a1 $\in E_i$;
                $E_i \leftarrow E_i$ - A1;
            LOOP: while $E_i \neq \phi$ do
                begin
                    while $E_i \neq \phi$ do
                        begin
                            A2 $\leftarrow$ a2 | a2 $\in E_i$;
                            $E_i \leftarrow E_i$ - A2;
                            if collapsible (A1,A2)
                                then
                                    begin
                                        $E_i \leftarrow E_i \cup$ collapse(A1,A2);
                                        leave LOOP;
                                        comment put collapsed result back
                                                to try to collapse it more;
                                    end
                                else
                                    E2 $\leftarrow$ E2 $\cup$ A2;
                        end;
                    comment we have exhausted the set
                            without collapsing A1 with anything;
                    RESULT $\leftarrow$ RESULT $\cup$ A1;
                end; comment end of LOOP;
            end;
        if E2 $\neq \phi$
            then
                return RESULT;
        return RESULT $\cup$ collapse.attributes (E2);
    end;
```

Two attribute sets are collapsible iff they differ in no more than one attribute-value pair, or formally:

$$\text{collapsible } (A_1, A_2) \text{ iff } A_1 = A_2 \lor (\exists \langle N_i, V_i \rangle \in A_1 \land \exists \langle N_i, V_i' \rangle \in A_2$$

$$\text{such that } \forall \langle N_j, V_j \rangle \in A_1 \land \forall \langle N_j, V_j' \rangle \in A_2, \text{ if } j \neq i \text{ then } V_j = V_j')$$

and where the collapse is defined to be the union of the values for the attribute which is different.

$$\text{collapse}(A1, A2) = A1 \cup A2$$

where we define "$\cup$" over attribute sets as a union over the attribute-value pairs, and the union over attribute-value pairs as a union over the attributes. As in the partial ordering operator, $\leq$, the context will always make clear which "union" we are performing:

$$A \cup B = \{ \langle N_i, V_i \cup V_j \rangle \mid \langle N_i, V_i \rangle \in A \wedge \langle N_i, V_j \rangle \in B\}$$

Note that this has also specified the $\cup$ operation for attribute-value pairs, i.e.

$$\langle N, v_1 \rangle \cup \langle N, v_2 \rangle = \langle N, v_1 \cup v_2 \rangle$$

Note that $\cup$ is not defined on attribute-value pairs if the attribute names differ. The operation $\cup$ on attribute values is the ordinary set union operator.

Language transformations

Language transformations represent the machine-independent transformations of the intermediate representation. Concepts such as commutativity or associativity are handled by language transformations.

In the structure of the BLISS/11 compiler (see Figure 3, page 28), the first three phases of lexical, syntactic, and flow analysis (known collectively as LEXSYNFLO) detect feasible global optimizations. It is the responsibility of later phases to decide which of the feasible optimizations are actually desirable. Given a tree or dag representation of the program, the use of a feasible optimization represents a change in the shape of the tree. If we wish to discover all possible M-operator sequences which evaluate a node, we must consider all possible representations of that node, given that all possible feasible optimizations are to be taken into account.

The use of language transformations allows us to generate the set of trees equivalent under feasible global optimizations.

There are various formalisms and techniques for using language transformations. One such set is given by Beatty [Bea72], and another by Sethi and Ullman [SU70]. The application of the cited techniques can be proven formally to minimize some cost function, such as register usage or memory accesses. The choice of any given technique depends upon the choice of a cost function---for example, if the only cost function involved was the minimization of memory references within very local scope, one particular technique may be chosen over another. Unfortunately, when faced with a "real" machine (which usually cannot be modeled in a simple manner) the problem becomes more complex. Idiosyncrasies of the machine _must_ be considered, not matter how strange. The use of global cost functions, such as complex register allocation schemes, introduces another level of complexity. An algorithm which produces the minimum number of memory references and uses the minimum number of registers may generate unusable code if the global register allocation strategy can only provide (at most) fewer registers than the minimum needed. In such a case, locally suboptimal code may produce a shorter or faster overall program.

The representation of language transformations chosen in the implementation was one sufficient to demonstrate that the use of language transformations is necessary to produce optimal code. Using such transformations, several equivalent code sequences were discovered which would not have otherwise been detected. It was also obvious that a more elaborate set of transformations would be necessary in a fully operational system.

Each language transformation is represented by a triple

$$\mathfrak{L} = < I, O, P>$$

where I is an Input pattern, O is an Output pattern, and P is a set of Preconditions which indicate when the application is feasible. Thus a language transformation which represents commutativity, (A+B)≡(B+A), would be represented by the triple < (A+B), (B+A), comm((A+B))>, indicating that for an input pattern (A+B) the output pattern (a new tree) is (B+A), and this transformation may be applied if the top node of the pattern (in this case, the + node) is commutative.

In specifying the patterns it is necessary to indicate whether or not the terminal nodes in the pattern must match terminal nodes in the tree being considered, or if they are permitted to match arbitrary subtrees. As a notational convenience we allow any terminal beginning with the symbol "S" to match an arbitrary subtree, and any terminal node with any other name is restricted to matching a terminal node[1].

There is no explicit commitment to how the predicates of the language transformations are implemented. In practice, they may be represented in the same manner as attributes (as they are in the BLISS/11 compiler). The fact that the same representation may be chosen internally should not cause any confusion. The two concepts are, in fact, conceptually disjoint in our model of code generation.

The language transformations and attribute transformations overlap at one point: the concept of expected sign. The expected sign of a result is both a language-related concept and a machine-related concept. Thus we find a language transformation of the form:

< (-S), (S[$SIGN OPPOSITE]), (value(S)≠-∞)>

---

1 The very simple pattern matcher implemented does not allow for complex pattern specification, so that separate patterns are required for commutativity of addition and multiplication, rather than a more general specification, such as < (A{+,x}B), (B{op}A), comm((A{op}B))>. This restriction had no significant impact upon the research, but might not be desirable in a production system.

Applications of language transformations usually do not converge. They will tend to either oscillate among a set of trees or to diverge. It is therefore necessary to decide when to cease application of language transformations. In our implementation, we chose to apply each transformation only once. Thus it was necessary to construct language transformations of the form:

<(S1+S2), (S1[$SIGN OPPOSITE] +[$SIGN OPPOSITE] S2[$SIGN OPPOSITE], (value(S1)$\neq -\infty \wedge$ value(S2)$\neq -\infty$)>

which would be equivalent to the application of the following transformations:

(i): <(S1+S2), -((-S1)+(-S2)), ...>
(ii): <-S1, S1[$SIGN OPPOSITE], ...>
(iii): apply (ii) again
(iv): apply (ii) again

The precise choice of language transformations can have a profound influence on the code generated. Although we have chosen a few that appear to be highly effective, there is no reason to believe that this set is complete. The choice was based upon experience, and upon observed behavior of the system while exploring various alternatives.

Search methods

The search technique consists of determining, for each node of a tree, all possible code sequences which could evaluate the operator at that node, and their associated costs. Some pruning techniques can be applied to the search in order to reduce the number of paths which must be explored, but a discussion of these will be deferred until the basic search technique is presented.

An underlying assumption is that we will search all possible paths in order to determine all possible code sequences. Although this can become expensive in terms of machine time, ideally it will be done only once per compiler. The results should also be more comprehensive (and more correct!) than the traditional manual generation of code sequences.

The search uses the idea of a difference operator, similar to that described by Ernst and Newell for GPS [EN69]. As we travel down the tree, we find, for each L-operator node, a set of M-operators (code templates) which will evaluate it. However, these code templates are partial functions, both over the domain of values they can operate upon and the domain of machine locations upon which they can operate. If we ignore the value domain (which can only be determined by the compiler we produce, for the specific program it is compiling) then we must map the locations of the operands into the domain upon which the M-operator can act. We represent the location domain as an attribute set.

The representation as attribute sets allows us to extend the idea of the location domain to cover more general attributes of an operand. Thus the expected sign, for example, becomes one of the attributes which we can use in determining the applicability of an M-operator sequence. By considering more general attributes of the operands we have available much more information which we can use to perform optimizations.

At any node which represents an n-ary L-operator we have n subtrees representing the operands. We apply a retrieval operator to the data base and obtain a list of all possible M-operator sequences which can be used to evaluate it. For each subtree we generate the preferred attribute set (PAS) as described earlier, and pass it down the tree. Upon return, we are presented with a set of M-operators, each of which represents the evaluation of the subtree, and each of which possesses an attribute set describing the result of this evaluation. There is guaranteed to be a null difference between each result attribute set and at least one member of the PAS[1]. When we have completely evaluated all the subtrees, the tree will now correspond to an n-ary L-operator and n sets of possible subtrees. If we call each set of subtrees T_i, then we can designate the size of each set as $|T_i|$. We can then form a new set $\mathfrak{T}$ by selecting all possible combinations of subtrees. The size of this set is $|\mathfrak{T}|=|T_1|*|T_2|*...*|T_n|$. For each M-operator M in $\mathfrak{M}$ we then check to see if there are any trees in $\mathfrak{T}$ whose subtree results satisfy the domain requirements of M. We then form the set which consists of all trees which represent the evaluation of the L-operator, including the attribute set which represents the result of the evaluation, and pass it back to the parent node. When we reach the root of the tree we have determined all possible code sequences which could evaluate the tree.

The selection of T-operators is determined in the current implementation by use

---

1 If this condition cannot be met, the search reports failure and is aborted to some higher level.

of a GPS-like difference operator, δ. The implementation and definition of δ depends upon how a particular implementation chooses to perform its search and what it requires to direct that search. The current implementation uses a very straightforward difference operator, which consists of matching corresponding value sets for each attribute name in the current-state description and the goal-state description. For each attribute-value pair in the current state description that is not satisfied in the goal-state description we produce a value which "describes" the difference. For example, if the value of $SIGN in the current state description is "+" and in the goal state description is "-", then the description of that difference is "OPPOSITE".

Note that the name chosen to describe a difference is arbitrary; any unique string of symbols would suffice. However, it is convenient to use symbols whose representation has some mnemonic value. Note that the same designator may be used as was used to name an active codepiece (see page 64). The use of a name is identical to using the explicit difference, i.e., "OPPOSITE" is equivalent to the explicit value set {'+' → '-', '-' → '+'}.

In order to present this more formally we must define some operators and explain the representations used. The basic operator is the <u>difference operator</u>, δ. To define δ we first define a simple operator, δ':

$$\delta'(C,G) = \{ \langle N, V\rangle \mid V \neq \phi, \text{ where } V = \{ y \to v_2 \mid \langle N, v_1\rangle \in \mathcal{C}(C) \wedge \langle N, v_2\rangle \in \mathcal{C}(G), y \in v_1 \wedge y \notin v_2\} \}$$

In the above definition C represents the current attribute set and G represents the goal attribute set.

The operator $\mathfrak{\delta}$ is only a slight extension to $\mathfrak{\delta}'$, and allows certain transformations to be named symbolically[1].

The data base, $\mathfrak{D}$, consists of a set of T-operators, as described on page 61. T-operators are retrieved by use of the retrieval operator, $\mathfrak{R}$, which is defined in terms of two sub-operators, $\mathfrak{R}_{lang}$ and $\mathfrak{R}_{att}$. The operator $\mathfrak{K}^*$, when applied to a T-operator, gives the "key" part of the operator, i.e., a member of the set $\mathfrak{K}$. $\mathfrak{R}_{lang}(OP)$ is used to retrieve T-operators which evaluate the L-operator OP, and is defined as

$$\mathfrak{R}_{lang}(OP) = \{ D \mid D \in \mathfrak{D} \wedge \mathfrak{K}^*: D = OP\}$$

and $\mathfrak{R}_{att}(AS)$ for an attribute set AS is defined as

$$\mathfrak{R}_{att}(AS) = \{ D \mid D \in \mathfrak{D} \wedge AS \leq \mathfrak{K}^*: D\}$$

1 In a manner similar to the use of "active" parts in a result pattern (page 64). For example, if the difference is in the $SIGN attribute, $\mathfrak{\delta}$ would indicate that a sign change is required by returning the value "OPPOSITE". The set of such differences is given as a set of triples, $\mathfrak{P} = \{n, x, v\}$, such that the first element of every triple is an attribute name n, the second element x is of the form $V_i \to V_j$, where V_i and V_j are allowable members of the set $\mathcal{V}_n$, (i not necessarily distinct from j), and the third element v is some new value, a member of the set $\mathfrak{E}_n$ (described on page 64). Thus, for the set $\mathcal{V}_{sign}=\{+\ -\}$, we have :

$\mathfrak{P}$ = { <$SIGN, + → -, OPPOSITE>
<$SIGN, - → +, OPPOSITE>
<$SIGN, - → -, SAME>
<$SIGN, + → +, SAME> }

The choice of names "OPPOSITE" and "SAME" is arbitrary, but the obvious mnemonic value of these names influenced their choice. The actual construction of the set $\mathfrak{P}$ would ultimately be performed by whatever system produced T-operators from a machine description; this set would be very large, because the set of possible transformations would be large. We have chosen a minimum set here for our example. The automatic construction of a set of T-operators is a major research project outside the scope of this thesis.

Chapter III

The Template-generator System

Introduction

Using the theory and notation evolved in the previous sections, a prototype system was constructed. The system was implemented in several ways, with the latest implementation written in LISP. It contains several major components: a parser which converts expressions into the desired internal representation; an "unparser" which converts the internal representation to a printable format (such as a tree or a string); the searcher, which given a goal and appropriate information about the allowable transformations will attempt to find the code sequence; and support code, such as tracing functions, interactive debugging facilities, file support, and several other facilities which were though necessary to make the LISP system habitable.

The implementation has been parameterized in such a way that the introduction of a new machine structure is a relatively simple task; all of the machine-dependent information has been isolated into a few "setup" functions for each type of machine.

In the next section, the details of the external (implementation) representation will be given. A machine (the Digital Equipment Corporation PDP-10) is presented, and a sample attribute set is constructed for it. Using this attribute set, we then construct a simple data base for some instructions on the PDP-10. A typical search is outlined, showing the results obtained from the original data base; the data base is then augmented and the results of this augmentation are shown.

External representation

We associate a set of attributes with every node of the tree, whether or not we presume that any M-operator is to be associated with the L-operator of that node. We also associate a set of attributes with the leaves of the tree, which describe the properties of the names associated with those leaves.

Although the symbols at the leaves of the tree have no significance to the pattern matcher, it is convenient for the user of the input language to attach some mnemonic significance to these tokens. Thus an operation which adds the contents of a register to the contents of memory could be expressed as (.X[$LOC REG $SIGN +] + .Y[$LOC MEM $SIGN +]) but it is more obvious and more concise to state (.R+.M). We thus attach attribute sets to both the external token and the dereferenced token; the current system parameterizes these in a very simple way in the parser[1]. The parser will also cause any attributes supplied explicitly by the user to override any default values.

The external representation of expressions for the current implementation is described informally here; for a more formal definition see appendix A. Identifiers are conventional, and the operators are likewise the usual +, -, *, /, and reserved word operators AND, OR, NOT, LEQ, LSS, GEQ, GTR, EQL, NEQ, MOD, XOR, LSH (logical shift) and the assignment operator, ←. The common IF-THEN-ELSE and WHILE-DO constructs

1 By use of the property indicators ATTRIBUTES and DOTATTRIBUTES on the property lists for the leaf names. Only the first character of the name is significant in locating the properties; thus M, M1, and MUMBLE are all equivalent (and happen to indicate memory locations). The complete description of the abbreviations used and their meanings for a specific application are given on page 87.

are also included, and are assumed to be expressions as in BLISS. Operations at the same priority level are always left associative, so multiple assignment such as A←B←C is interpreted as (A←B)←C, rather than the expected A←(B←C)[1].

The external representation of an attribute set is a bracketed list of attribute-value pairs. There is no explicit pairing delimiter. Everything between one attribute name and the next is considered to be the attribute value. This is a simple decision, since attribute names are distinguished symbols; each begins with the character "$". Experience indicated that an explicit pairing delimiter, although "formally" correct, merely cluttered up the representation to the point of illegibility.

An attribute set may be associated with any symbol in the expression simply by writing it after that symbol. It is then attached to the same node in the tree as the symbol. Examples are:

.R[$LOC LIT] + .[$LOC MEM] M

Which associates the pair <$LOC LIT> with the symbol R and the pair <$LOC MEM> with the immediately preceding . for this particular tree.

1 Please note that this is only a peculiarity of the particular syntax analyzer involved here and should in no way be construed as a commitment by the author.

A Sample Attribute Set

We will now illustrate the use of attributes with a concrete example. The set here is used to represent PDP-10 machine code [DEC72]. All examples given, until stated otherwise, are based upon the PDP-10 architecture.

Briefly, the PDP-10 is a rather conventional multiregister machine, with 16 "general purpose" registers which can be used as accumulators, floating point accumulators, index registers, or memory locations. These registers reside in locations 0-15 (decimal) of the address space, thus eliminating the need of special register-to-register instructions. The instruction code is highly symmetric; within a class of instructions, if one option exists (such as a test for equality), it is usually the case that all options exist (such as the other five relationals). Arithmetic and logical instructions can operate in several modes, such as R←(R op M), R←M←(R op M), M←(R op M), etc., allowing results to be developed in a register, in memory, or both simultaneously.

Control consists of both skip-type instructions and transfer-type instructions, several varieties of subroutine calling mechanism, including nested subroutine calls by use of a stack. One or more registers can be designated stack pointers, and there are instructions for pushing data onto and popping data from the stack; the stack pointer register is addressed explicitly in the push, pop, call, and return instructions. It should be pointed out that the PDP-10 is <u>not</u> a stack machine; it only allows data to be pushed onto and popped from the stack, as well as allowing subroutine calls and returns to store and use addresses on the stack. There are no stack operations, such as ADD, which implicitly pop the stack. Of course, since the stack pointers are general purpose registers, the register can be used to index onto the stack. As in many general register architectures, register 0 cannot be used as an index register. Certain instructions which allow an option of producing a result in a register can only use registers other than register 0 for the same reason: if the register field is zero, it implies that a register is not involved.

The entire address space of the PDP-10 is directly addressable by any instruction, i.e., no "base registers" are required to make the entire 256K address space available. Address decoding is completely consistent in all instructions, where an address consists of a 23-bit quantity in the low-order bits of an instruction: 18 bits of direct address, 4 bits of index register designation, and 1 bit which is the indirect addressing flag. When fetching an indirect address, the entire low-order 23 bits of the address retrieved are decoded, allowing infinite-depth indirection. Any subfield of a single 36-bit machine word can be accessed, for either storage or retrieval, by a construct known as a "byte pointer". A byte pointer contains a 23-bit address field (decoded as described), and two 6-bit values representing the position of the low-order bit of the subfield (relative to the low-order bit of the word) and the number of bits in the subfield.

All fixed-point values are represented as two's complement binary numbers; and floating point values have an 8-bit excess-128 exponent with a two's complement fraction (27 bits, with the sign bit the high-order bit of the word). In addition to memory and registers, there are several flags directly testable by the user for conditions such as overflow, and some other status bits which are used by the central processor. The remainder of the architecture, including relocation and I/O structure, is of no concern here.

For any architecture there are a set of locations in which data can be stored.

These locations may be main memory, auxiliary memory (scratchpad), registers, condition codes, program counter, etc. Some, or possibly all, of these locations are addressable from machine instructions; most architectures restrict addressability by requiring special instructions to access special locations. When it becomes necessary to use an intermediate result, it may be necessary to transfer from one location to another, in order to make it addressable.

Let us define

$\mathcal{N}$ = {$LOC $SIGN $ADDR $DT $CSE $PS}

These attribute names represent the Location, Sign, Addressability, Destructability, Reusability and Bitwise Position and Size of the results we wish to deal with. The value sets are indexed by these names and we will designate the value set of some name i by the notation $\mathcal{V}_i$. We can now define the value sets:

$\mathcal{V}_{loc}$ = {REG0 REGN0 MEM PC OVF CRY0 CRY1 FOVF LIT}

REG0 is register 0; REGN0 is any register other than register 0[1]. MEM is memory, PC is the program counter, and {OVF CRY0 CRY1 FOVF} refer to the four directly testable flag registers: fixed point overflow, two types of carry, and floating overflow. LIT is any value which can be stored as a constant in an <u>immediate</u> machine instruction. In addition, we will define an abbreviation REG="REG0, REGN0".

$\mathcal{V}_{sign}$ = {TRUE, COMP, +, -}

These attributes define the condition of the result <u>relative to the desired</u>

1 This distinction is necessary since register 0 cannot participate in certain operations or functions, e.g., indexing.

condition. The choice of this set is made by inspection of the PDP-10 instruction set and study of existing compiler designs [Wu70, KKR65]. Examination shows that it is often possible to produce a result at a node which is in the complement condition of the desired result, and produce it at lower cost than the uncomplemented result. Presumably it can be complemented during a later evaluation, if required, but such an operation may not be required. For example, in computing -.A*-.B it is not necessary to actually compute the negative of either operand in order to obtain a correct result. "TRUE" and "COMP" deal with logical (36-bit) results while "+" and "-" deal with signed (35-bit plus sign) results.

Note that this attribute deals with those aspects of data representable by a unary complement operator, i.e., an operator which when applied twice to the same data has no net effect [Fra70]. Note also that it is not always true that -(-.A)=.A. In a two's complement binary representation such as the PDP-10 uses there is one value (the largest negative number) to which this does not apply. This issue will be discussed in more detail later. If one were to ignore certain finite-precision issues dealing with floating-point representations, it would be desirable to include another attribute indicating whether the result were the true or inverted result relative to division.

$$\mathcal{V}_{addr} = \{\text{REG, MEM, EA, BYTE, FLAGS}\}$$

This attribute defines the addressability of results. It is convenient, although perhaps misleading, to use the same symbols here as in the LOC attribute. REG means that the result can be addressed by the register field of most instructions. MEM means that it can be accessed by an 18-bit address which points into memory (which thus includes the registers). EA means that it can be addressed by an effective address calculation performed during the instruction fetch cycle. BYTE means that a PDP-10 byte pointer can access the result. FLAGS means that one of the instructions which interrogates the flag register can access the result.

$$\mathcal{V}_{dt} = \{\text{YES, NO}\}$$

This attribute is related to the language-level concept of common subexpressions, and indicates whether or not the result will be required later and therefore must be saved. It is important at this level since most M-operators destroy one of their operands. The concept of the "destroyable temporary" also applies to situations where user-defined variables are involved, such as in the expression (A←.A+1), where the computation may be done by an increment instruction which adds 1 to A, providing the value of A is not otherwise required.

$$\mathcal{V}_{cse} = \{\text{YES DONE NO}\}$$

This indicates whether or not a node is a common subexpression, and if so whether or not its result has been computed. (YES implies ~DONE; DONE implies YES). This is useful in determining whether or not code must be produced to evaluate a node; if the attribute is YES then code must be produced to evaluate the node. The result of executing this code is to produce a node whose attribute is DONE.

$$\mathcal{V}_{ps} = \{ \langle p,s \rangle \mid 0 \le p \le 36, 0 \le s \le 36, 0 \le p+s \le 36\}$$

This attribute represents the <u>position</u> and <u>size</u> of an operand within a data word. Note that although these names follow directly from the PDP-10 definitions, they in fact are machine independent. For a discussion of this machine independence, which is also radix independence, see Knuth's MIX computer [Kn68, p.120ff]. We will follow the PDP-10 and BLISS conventions that the position is the number of bits (or basic information units, in general) from the right end of the data word and the size is the width of the field in the same information units. Besides the obvious use in computing shift instructions, there are some other uses relating to the finite precision available in the machine representation which are handled by this attribute[1].

In the implementation we have provided for a shorthand for defining the attributes associated with a symbol. We will now define some of these abbreviations here, and then use them implicitly in subsequent examples. Note that when we must distinguish between two nodes with the same desired attributes, we will append numbers to the symbols given here, e.g. M1, M2, etc.

M	[$LOC LIT $ADDR LIT $DT NO $CSE NO $SIGN TRUE $PS <0,18>]
.M	[$ADDR MEM $SIGN (TRUE +)]

> Note that the other attributes of the .M node, as in all other dereferenced nodes, are undefined unless explicitly stated here. These are considered to be either the empty set or the universal closure set $\mathcal{C}_i$ unless changed during the evaluation of the tree.

R	[$LOC LIT $ADDR LIT $DT NO $CSE NO $SIGN TRUE $PS <0,4>]
.R	[$ADDR REG $SIGN (TRUE +)]
EA	[$LOC LIT $ADDR LIT $DT NO $CSE NO $SIGN TRUE $PS <0,18>]
.EA	[$ADDR EA $SIGN (TRUE +)]
X	[$LOC LIT $ADDR LIT $DT NO $CSE NO $SIGN TRUE $PS <0,18>]
.X	[$ADDR (REG MEM) $SIGN (TRUE +)]
PC	[$LOC PC $ADDR PC $DT NO $CSE NO $SIGN TRUE $PS <0,18>]

1 Note than in the sample printouts which appear later that the $PS property is represented by <u>two</u> properties, $POS and $SIZ. This was a concession to an implementation quirk, and does not affect the formal definition. A formal definition of $PS as two attributes is clumsy and serves no useful purpose.

Given these abbreviations, we can now explore some of the instruction set of the PDP-10. By way of examples, consider the following PDP-10 instructions:

Transformation	M-operator	Cost[1]
R→M	MOVEM R,M	ref: 2; size: 1
M→R	MOVE R,M	ref: 2; size: 1
PC→R	JSA R,.+1	ref: 1; size: 1
LIT→EA	none	ref: 0; size: 0
	(Accomplished by a T-operator)	
EA→PC	JRST EA	ref: 1; size: 1
EA→PC, PC→R	JSA R,EA	ref: 1; size: 1

1 In this greatly simplified cost function we treat all register accesses as 0 cost and all memory accesses as unit cost. Each instruction fetch cycle is assumed to require one memory access, which ignores indirect addressing. The real cost function which an actual compiler would use would be far more complex.

A Sample Data Base

We will now illustrate the use of the system by constructing a small data base and then using it to construct some machine-code templates. We will first define the semantics of the terminal symbols, then construct the T-operators. The machine used in this example is the PDP-10 [Dec72]. In this example we do not use the full set of attributes for the PDP-10 as defined on page 84, but use a subset. This was done because our examples do not require the complete set of attributes, and to include them would only make it more difficult for the reader to readily understand what is happening.

```
 R   [$LOC LIT $PS <0,18> $SIGN (+ TRUE) $ADDR LIT $CSE NO $DT NO]
.R   [$LOC REG $PS <0,36> $SIGN (TRUE +) $ADDR REG $CSE NO $DT YES]
 M   [$LOC LIT $PS <0,18> $SIGN (TRUE +) $ADDR LIT $CSE NO $DT NO]
.M   [$LOC (REG MEM) $PS <0,36> $SIGN (TRUE +) $ADDR EA $CSE NO $DT YES]
 X   [$LOC LIT $PS <0,18> $SIGN (TRUE +) $ADDR LIT $CSE NO $DT NO]
```

We define two more special values for the attribute $SIGN. These special values can only be used in the result part of a T-operator, and instead of representing static values they represent either a variable in the pattern (in the pattern part) or a transformation to be performed on the actual input node by the compiler (in the result part). The ability to use and name such special values is described on page 64 for result parts and page 77 for patterns. The two values are OPPOSITE, which in the pattern part indicates that a sign change is desired and in the result part indicates that a sign change has occurred, and SAME, which indicates a sign retention is desired or effected. A trivial extension of the example given on page 78 to include the attributes TRUE and COMP would suffice here.

The set of T-operators described here is quite small, and is the minimum set required for an interesting example without exceeding reasonable limits on the size of the example or the patience of the reader.

Key	+
Pattern	.R+.EA
Result	.R
Preconditions	None
Postconditions	None[1]
Code	<ADD R,EA>
Cost	Ref: 2; Size: 1

Key	+
Pattern	.R+.EA[$SIGN -]
Result	.R
Preconditions	value(.EA)≠-∞
Postconditions	None
Code	<SUB R,EA>
Cost	Ref: 2; Size: 1

Key	[$LOC MEM→REG $SIGN SAME]
Pattern	.EA
Result	.R
Preconditions	None
Postconditions	None
Code	<MOVE R,EA>
Cost	Ref: 2; Size: 1

Key	[$LOG MEM→REG $SIGN OPPOSITE]
Pattern	.EA
Result	.R[$SIGN OPPOSITE]
Preconditions	value(.EA)≠-∞
Postconditions	None
Code	<MOVN R,EA>
Cost	Ref: 2; Size: 1

Key	[$LOC LIT→EA $SIGN SAME]
Pattern	X
Result	EA
Preconditions	None
Postconditions	None
Code	<>
Cost	Ref: 0; Size: 0

1 There is an implicit postcondition to every operator which states that the result (such as .R) represents the evaluation of the tree node.

The choice of representing the operands by "EA" is based on the fact that the PDP-10 can use the effective-address calculation to determine the location of a data word. The typical operand in a fixed location (such as an "OWN" variable in the BLISS or Algol sense) has an address which is represented by an 18-bit literal address, while an operand on the execution-time stack is represented by an offset from a stack pointer register. The "ADD" instruction will work equally well with either type of operand, so the more general representation of an effective address is used. However, this does require that an operand be converted to an effective address. We wish to have a transformation which converts a literal or stack-relative address into an effective address. We locate such transformations by applying the difference operator δ to the current state of an operand attribute set and the desired state of an operand attribute set (how we obtain the desired state is discussed later). We then obtain a difference which we use to search for a difference-reduction operator. Difference-reduction operators are indexed in terms of attribute sets, where an operator O_i is considered applicable to reduce a difference D_j iff $D_j \leq \text{index}(O_i)$.

This explains the appearance of the T-operator with the key [$LOC LIT→EA $SIGN SAME]. Any time the difference operator requires an effective address from a literal, this is one of the possible choices. Note that converting a literal to an effective address is a trivial operation, and in practice consists of putting the literal value into the 18-bit address field. In other architectures, for example that of the IBM/360, addresses are computed as offsets from base registers. The literals are restricted to the range 0 through 4095. If an arbitrary address has a base register available, then the conversion from address to effective address consists of specifying a base register and an offset, which in most instructions incurs no cost. However, if no base register is available, it is necessary to use any number of a set of techniques for computing the address, i.e., changing a machine address to an effective address. These will incur various costs in terms of code size and/or number of memory accesses. Thus the appearance of zero-cost T-operators such as the one to convert a literal to an effective address are not incidental, but fundamental to the proper formulation of the data base.

Example: a typical search

This section presents one example in depth in order to examine the system in detail. A complete printout of the example described here may be found in Appendix B.

We are given the tree:

.M1+.M2

The data base is searched for a set of T-operators which could evaluate the tree, and we obtain:

ADD R,EA

SUB R,EA

We then form the preferred-attribute set (PAS) for each of the subtrees, based on the forms acceptable to the machine instructions, and we obtain

. [$LOC REG] R + .[$LOC EA $SIGN (+ -)]

i.e., the PAS for the left subtree is the same as that for .R, and for the right subtree it is [$LOC EA $SIGN (+ -)].

The program then uses this pseudo-tree in its tree-search. It performs an endorder walk in parallel on the original tree and the PAS tree. When it encounters the node .M1, it compares the node to the corresponding node of the PAS tree, an discovers that they are different; specifically, we require the operand to have the attribute $LOC REG.

The program then attempts to transform the current node into one which satisfies the criterion by using the definition of the goal state to search the data base for a T-operator. In this manner, it finds two T-operators which will move something into a register, and which have associated M-operators, specifically:

```
1) [$LOC MEM→REG $SIGN SAME]
        MOVE R,EA
2) [$LOC MEM→REG $SIGN OPPOSITE]
        MOVN R,EA
```

The data base is searched using the ≤ operator to test the T-operators against the goal state; any T-operator whose key K_i satisfies the goal G_i, defined as $G_i \leq K_i$, is acceptable. Thus we obtain two T-operators, both of which leave their result in a register, but which have different effects upon the sign. These T-operators are then checked to see if they can operate upon the node in the tree; both in fact are accepted.

As we search the right subtree, we find a similar difference in the actual operands and desired operands. The PAS indicates that we can act upon any operand which has the attribute [$LOC EA]. However, the actual operand (M2) has the attribute [$LOC LITERAL]. We use the difference in the same manner to locate a T-operator which will transform the attributes. In this case, we obtain the single T-operator:

```
[$LOC LITERAL→EA $SIGN SAME]
      <nil>
```

Note that there is no M-operator associated with this transformation. This is because a literal can be changed to an effective address simply by having the compiler place its value in the effective address field of an instruction, and this incurs no execution time cost.

We have now generated sets of subnodes which represent evaluations of the subtrees, i.e.,

```
           +
          / \
         /   \
  { . .- } { . }
    | |      |
    R R      EA
```

If we take all possible combinations of left and right subnodes, we would have two trees, representing

```
.R + .EA
. [$SIGN -] R + .EA
```

Every time we attempt to process a tree, we also attempt to process any trees equivalent to it under the language axioms. In this example we have only one axiom, which states

$$-(-A + -B) \equiv (A + B)$$

We apply this axiom to our candidates, and obtain the following set of equivalent trees:

```
.R + .EA

. [$SIGN -] R + .EA

        (the two original trees obtained from the search)

.R + .[$SIGN -] EA

.[$SIGN -] R + . [$SIGN -] EA
```

This set of trees is then compared against the code sequences possible, and the following two complete code sequences are thus obtained:

```
MOVE R,M1
ADD  R,M2
```

```
MOVN R,M1
SUB  R,M2
```

Note that the latter case produces a result with the opposite sign than that expected. This information could then be passed up the tree to a higher node.

The result of this search has produced two templates which can accomplish an "ADD" operation. Given the data base, these are the only two possibilities. Some of the results obtained with more complex data bases have produced more alternatives.

Each extension of the data base has provided more possibilities for code sequences. In the data base given on page 90, it is impossible to generate code for adding the contents of a subfield of one memory location to the contents of another location. By adding the bit-field extraction operator, Load Byte (LDB), we can perform this operation. The PDP-10 requires the specification of a 36-bit word which is the "byte pointer", and is used as an indirect reference to the word containing the byte. A byte pointer defines a position of the byte, in bits from the right end of the word, and a size of the byte in bits. The low-order 23 bits of the byte pointer are interpreted in the same way as a machine instruction address. Thus the instruction:

```
LDB R, [BYTEPOINTER POS, SIZ, ADDR]
```

will move into register R the subfield of the word at ADDR described by POS, SIZ. The appearance of the BYTEPOINTER pseudo-op in square brackets causes the assembler to treat it as a literal and place it in the literal pool[1]. This operation is the general implementation of the BLISS subfield operation, ".ADDR<POS,SIZ>". If we are given the tree:

[1] Those familiar with the PDP-10 assembler will recognize the liberties being taken for the sake of exposition. The BYTEPOINTER pseudo-op does not exist, but the more incomprehensible POINT pseudo-op is its realization.

Example: a typical search

.R + .M2<18,18>

we obtain the code sequence:

```
LDB     r,[BYTEPOINTER 18, 18, M2]
ADD     R,r
```

which, in its most general application also allows us to use the sequence LDB r,[BYTEPOINTER 0,36,M] in place of a MOVE instruction. Although these two sequences are equivalent, the lower-cost MOVE instruction would be used in preference.

The possibility of a lower-cost alternative code sequence can be shown by adding the halfword-move instruction HLRZ to the data base. The HLRZ instruction moves the left half of the memory location to the right half of the register, and zeroes the remaining half of the register. Thus, our original tree of .R + .M2<18,18> can also be evaluated by the code sequence:

```
HLRZ    r,M2
ADD     R,r
```

The PDP-11 is a 16-bit computer with several addressing modes. These modes allow computations to be performed register-to-register, register-to-memory, memory-to-register and memory-to-memory (there are other options, but they are not relevant here). Thus it is possible in the PDP-11 to perform computations which may not involve the use of any intermediate registers. However, for each operand which requires a memory address, an additional 16 bits is required in the instruction. A memory-to-memory instruction requires 48 bits (three words), while a register-to-register instruction requires only 16 bits (one word). Operations are of the form "OP src dest", where "src" is the source operand and "dest" is the destination operand.

Optimization of code in the PDP-11 is therefore complicated by the tradeoffs in time and space between using registers (of which only 6 are available in most cases) for operands and leaving the operands in memory.

Using a short data base for the PDP-11, which described three basic types of ADD instruction (add memory to memory, add register to memory, add memory to register), and a single instruction which would move the contents of memory to a register, the tree .M1 + .M2 was evaluated by three different code sequences, with approximate costs as indicated:

Instruction	Ref	Size
MOV M1,R1	3	2
MOV M2,R2	3	2
ADD R2,R1	1	1
	----	----
	7	5
MOV M1,R	3	2
ADD M2,R	3	2
	----	----
	6	4
ADD M2,M1	6	3
	----	----
	6	3

In this case, the choice the compiler would make would be influenced by factors not known until the actual program was being compiled. For example, the direct memory-to-memory addition is the shortest sequence, but it presumes that the contents of location M1 can be destroyed by this operation [$DT YES]. This fact cannot be determined until a program is compiled, and the global properties about the locations are fixed.

Returning to the PDP-10, we add some machine operations for adding constants.

A constant whose value is less than $2^{18}-1$ can be added by use of an immediate instruction, where the actual value of the constant occupies the address field. If we consider the example of .R + k, for k a constant, then that tree can be evaluated with an instruction of the form ADDI R,k. However, because of the nature of index registers on the PDP-10, if we know that the result must be less than $2^{18}-1$ then we can use indexing to perform the addition[1]. The tree .(.R + k) could be evaluated by either of these sequences:

```
ADDI    R,k
MOVE    r,0(R)

MOVE    r,k(R)
```

Our system discovers that for this case constant arithmetic can be performed by indexing, at (nominally) zero cost; the result of evaluating .R + k is indicated as either the ADDI instruction (whose result is a 36-bit value) or by indexing (whose result is an 18-bit value).

Template output

The actual templates generated are very large list structures. For an example of the actual templates for the first example, see the structure named "*CODESET" given in Appendix B on pp. 128-136. From this structure all the information necessary to construct the T-operators may be obtained.

1 If we had a compiler which accepted assertions about the values taken on by a variable we could perform this optimization any time the result was known to be a positive integer whose value was less than $2^{18}-1$. Currently the only time the compiler knows that this assertion is true is when the value is used as an address.

Chapter IV

Conclusion

Summary

This thesis has been intended as a contribution to the technology of compiler construction. Because of the breadth and complexity of the task, we have made certain assumptions and restricted the scope in certain ways. We have assumed that we will be working within the framework of a compiler-compiler system; the flexibility of such an approach has been proven, and it is now necessary to provide the technology which will remedy a serious deficiency in most compiler-compilers, namely the lack of adequate techniques for describing code generation. We have also presumed a specific model of the compiler which such a system would produce, specifically, the basic structure of the BLISS/11 compiler. We have then focussed on one particular module of the BLISS/11 compiler, DELAY. This compiler model was chosen primarily because it performs substantially better than the classical compiler model in the area of optimizing code; the issues are largely separable from those of the external syntax.

Ultimately, a system such as we have modelled here will be able to produce a DELAY module for a compiler. We have not limited the class of languages to BLISS, but the general class of algebraic languages, including Algol, FORTRAN, and a large subset of PL/1. The direct output of a program such as the one developed here is <u>not</u> the DELAY module, but a set of information which could be used to construct a DELAY module. Depending upon the approach taken in developing the compiler-compiler, we forsee between one and three processing steps required to convert the template information into the actual code (or tables) of a DELAY module.

The major contribution of this thesis is the characterization of machine operators in terms of attribute sets, and the use of attribute sets in a search strategy which permits the discovery of code sequences using only the abstract concept of attributes. The formalization of the preferred-attribute set and its relation to attributes makes it possible to take advantage of the search strategy found highly effective in the BLISS/11 compiler.

Several examples were run to demonstrate the flexibility of this method. Although there remains a significant amount of work before an actual production version of this system is operational, and is integrated into a compiler-compiler system, that work represents a one-time investment. The result will be a system which makes it possible to produce high-quality compilers with much less effort than is currently expended.

Relationship to Automatic Programming

By this point the reader who is familiar with the ideas of "automatic programming" has undoubtedly seen the similarity between the methods used here and those used in automatic programming. Our goals are remarkably similar; consider this statement from Buchanan [Buch74]:

> The need for some automation in the task of software production is becoming increasingly clear. Systems are getting bigger and more complex which has caused maintenance cost to rise (it is now 50% of the programming budget). Software costs too much, isn't reliable, takes too long to develop and is difficult to modify or fix. Programming has not attained the maturity to develop standard engineering practices with their attendant reliability that other disciplines have. Research in automatic programming seeks to understand the nature of the task and thereby improve production.

One of the earliest attempts to automate a compiler was the "heuristic compiler" of Simon [Sim63]. Given a set of input-output relationships between data, the heuristic compiler produced a piece of IPL-V code (or, in our notation, a series of M-operators for an IPL-V machine) which processed data according to the desired relationship. The Heuristic Compiler was implemented (in one of its versions) in GPS [EN69]. In this model, the data base we use becomes the GPS "table of connections"---specifically, it indicates allowable transformations and possible difference-reducing operators. However, the requirement of <u>minimum</u> <u>cost</u> code means that searching for the first set of operators which satisfies the goal is not adequate;

we must prune the search using other criteria which are highly task-dependent. Although it would be possible to express our problem in GPS, the solutions would not be satisfactory unless GPS were modified.

Our model of the compiler-compiler and the compiler processes are very similar to those proposed by Buchanan [Buch74]. The differences tend to be in the representations chosen to represent input-output conditions (primarily represented as attribute sets), the form of indicating desired solutions (preferred-attribute sets), and the minimum-cost criterion. Many of the techniques used by Buchanan would be desirable in a production system, such as synthesis of conditional statements and loops. Loop synthesis, in particular, could be used to construct such M-operator sequences as multiple-bit shift operations on machines with only single-bit shift instructions.

One of the features of the automatic programming system described by Buchanan is the "program library"; once a program "A" with input predicates "P" and output predicates "Q" (represented as P{A}Q) has been generated, it may be "generalized" and stored in the program library. Generalization is described as a process by which a procedure declaration is created for a code segment (which is in Algol-60), and a set of goal, input, and output conditions are specified. If we call the goal conditions "retrieval keys and input patterns", and include cost data, then in fact the program library resembles the code template lists used by a compiler. We could then use such a system to construct more powerful templates, or complete subroutines (perhaps a machine-dependent I/O and device support package in assembly code would be a significant example).

The ability to add cost data to the program library is a significant advance. For example, a procedure to compute a simple function such as rounding a real result and converting it to an integer might, as a side effect, recursively compute Ackermann(5,5); since the internals of a procedure are invisible this might never be discovered. By adding cost data we can always choose the minimum-cost solution and determine if the minimum-cost solution is feasible. Note that the cost data may not be a constant; it may be a function of any or all of the input parameters, and the resultant cost function may thus be a parameterized function which requires additional information to produce an actual cost figure. Judicious choice of the parameters should produce lower and upper bounds on the computation.

Evaluation and retrospect

The ideas presented here form a basis for a system which automatically produces optimizing compilers. There are, of course, a number of unsolved problems; several of these are interesting research projects in their own right. The implementation of the system was intended to be a test bed on which to try out various ideas. Thus it has many limitations which preclude its use in a production environment. For example, it was coded in LISP[1], which provided a very nearly ideal environment for interactive development and deougging of the system, but which also incurs enormous overhead in space and time. Although the system could be compiled, the factor of 10-100 in speed improvement will probably not be sufficient to process a real machine description. Since it was intended at the outset that this implementation would be only a prototype, there are numerous deficiencies (or inefficiencies) in some of the internal algorithms. In most cases these deficiencies were left in because their removal, although it would produce a more aesthetically pleasing system, would not contribute to understanding the problems of code generation.

None of the limitations we encountered seem to be inherent in the basic ideas, only in one particular realization of them. However, for the benefit of those who may wish to pursue this research further, we would like to summarize some of the problems we encountered in implementing the ideas developed here.

The search strategy implemented needs to be more powerful. In particular, methods of reducing the total search space should be implemented. The concept of

1 The UCI extensions of Stanford LISP 1.6.

"theorem" is relevant, for example, in that once a certain search is performed the set of T-operators should be retained in such a way that any future attempts to perform the same search would only need to access the theorem, not "re-prove" it from the basic "axioms". This process of retaining knowledge is usually an important component of most artificial intelligence programs; we did not include it in our implementation because the effort of coding it would not be justified. It was clear from the observed performance of the system that such a component would be essential in other than a prototype system.

Faster methods of combining sets in order to test the validity of the results of subtree searches could have been implemented. In the current implementation it was expedient (from the programming viewpoint) to form the cross-product space of all the subtree searches, and then reject all those members of the cross-product which did not meet the requirements of the node. For the small examples for which this prototype system was intended that was satisfactory, but it would be far too expensive to employ a similar technique in a production system.

The concept of "semantically equivalent" sequences has not been implemented. Although precise determination of semantically equivalent code sequences is unsolvable [ASU70], there are many cases for which this is solvable. The rejection of all semantically equivalent sequences of equal or greater cost than the chosen one would not only reduce the amount of time required to discover the code templates, but would also reduce the amount of time required by the compiler to examine the alternatives for a given node.

Many minor problems arose during the implementation due to the fact that the system was a test bed; as our understanding of code generation evolved through experiments with the system, it likewise evolved to become more general and more powerful. The resultant system accurately follows the model of code generation presented here.

Future research

Many areas of future research remain. The most important area is the transformation of the templates generated by this type of system into a DELAY module. There are many techniques which could be used for this. Whether we use a "table driven" DELAY, where the code remains constant and the data changes, or a "decision table" generated DELAY, where the data is used to generate the code, are only two choices. The use of more sophisticated automatic programming techniques, such as the frame model of Buchanan, is another, not necessarily mutually exclusive possibility. Nonetheless, before an actual compiler can be built we must be able to produce a functioning DELAY module.

We also need to use more powerful artificial intelligence techniques to generate the templates. The construction of a production version of this system would involve the development of efficient methods of implementing the basic ideas presented here.

An actual compiler, or course, involves more than just the DELAY module; it includes everything from lexical analysis to formatting a file of relocatable code. We could use many standard methods of parsing the input, although new ones are still being developed. The discovery of common subexpressions and feasible global optimization strategies has been modelled, and that model has been realized in BLISS/11; however, the generation of a FLOWAN for an arbitrary language from the description of the ordering relations (such as those of Geschke [Ges72]) is still a manual process. The construction of modules such as TNBIND and FINAL must still be automated; partly because these modules represent significant amounts of time to implement, and partly because our model of DELAY is not compatible with the current implementations. The latter case implies that even a direct copy or a re-implementation of the TNBIND and FINAL models from the BLISS/11 compiler would not operate with the new DELAY.

The actual construction of a compiler using this model is still an open area for research. The model presented here requires that every possible tree which represents the evaluation of an expression be generated before the least-cost tree is discovered. Although it might be possible to use clever encoding techniques to reduce the exponential explosion in memory requirements, such a solution does not reduce the basic complexity of the exhaustive search. Like others faced with problems of exhaustive search, we would like to find a set of heuristics which reduce the search and do not unduly limit the solutions found.

A number of heuristics are possible; for example, a set of heuristics which would tend to generate the trees in ascending order of cost would require generating only one tree at a time. The construction of a compiler which uses heuristic methods to perform "nearly as well" as one which performs the exhaustive search is clearly an area of future research.

There are other, long-term, research problems which would contribute significantly to our goal of constructing correct compilers. The generation of a set of attributes and T-operators is certainly one of these. In the ideal model of automatic compiler production, only the machine description(s) and language description(s) are required; everything else is automatic to the generation of the final compiler. If we could generate the attributes and T-operators for the data base directly from a machine description we would have reduced a significant possibility of errors in the final compiler.

There will be a need for optimizing compilers as long as it is possible to benefit from them. Even a breakthrough in computer architecture which makes it difficult to produce <u>inefficient</u> code will not replace all the existing machines in the world. The reductions in resource utilization made possible by using an optimizing compiler are great enough that there will continue to be emphasis on them in the future. However, the cost and difficulty of constructing an optimizing compiler for a given language/machine combination is high enough that the investment is often not justifiable except by a manufacturer or software house. Increasing costs will make the investment even less feasible in the future. Thus, we must now begin to develop the tools necessary to automate the construction of compilers. This thesis is one contribution towards that goal.

BIBLIOGRAPHY

[AGG69] Arden, B. W., Galler, B. A., and Graham, R. M., "The MAD Definition Facility", in CACM 12, 8 (August 1969), pp 432-438

[Al71] Alsberg, P. A., OSL/2, An Operating System Language, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1971

[ASU70] Aho, A. V., Sethi, R., and Ullman, J. D., "A Formal Approach to Code Optimization", in SIGPLAN Notices 5, 7 (July 1970) pp 86-100

[Bai72] Baird, G. N., "The DOD COBOL Compiler validation system", Proceedings of the FJCC 1972, pp 819-827

[Bar73] Barbacci, M. R., A Comparison of Register Transfer Languages for Describing Computers and Digital Systems CMU CSD report, March 1973 (Revised May 1974)

[Bea72] Beatty, J. C., "An Axiomatic Approach to Code Optimization for Expressions", JACM 19, 4 (October 1972) pp 613-640

[BN71] Bell, C. G., and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971

[BS74] Barbacci, M. R. and Siewiorek, D. P., Some Aspects of the Symbolic Manipulation of Computer Descriptions, CMU CSD report, July 1974

[Buch74] Buchanan, J. R., A Study in Automatic Programming, CMU CSD report, May 1974

[CH74] Conradi, R. and Holager, P., MARY Textbook, Regnesentret ved Universitetet i Trondheim (RUNIT), July 1974

[CLE69] Carr, C. S., Luther, D. A., and Erdmann, S., The TREE-META Compiler-compiler System, Report No. TR 4-12, Computer Science Department, University of Utah, March 1969

[Cole74] Coleman, S. S., JANUS: A Universal Intermediate Language, Ph.D. Dissertation, Colorado University, May 1974

[Con58] Conway, M. E., "Proposal for an UNCOL", CACM 1, 10 (October 1958), pp 5-8

[CS70] Cocke, J., and Schwartz, J. T., Programming Languages and their Compilers, Preliminary Notes, Courant Institute of Mathematical Science, New York University, New York, April 1970

[Dah67] Dahl, O., et. al., Simula 67: Common Base Language, Norwegian Computing Center, University of Oslo, 1967

[DEC71] PDP-11 Processor Handbook, Digital Equipment Corp., 1971

[DEC72] DECsystem10 Assembly Language Handbook, second edition Digital Equipment Corporation, 1972

[Don73] Donegan, M. K., An Approach to the Automatic Generation of Code Generators, Ph.D. Thesis, Rice University

[EN69] Ernst, G. W. and Newell, A., GPS: A Case Study in Generality and Problem Solving, ACM Monograph Series, Academic Press, 1969

[ER70] Elson, M., and Rake, S. T., "Code-generation technique for large-language compilers", IBM Systems Journal 9, 3 (1970), pp 166-188

[ES70] Early, J. C., and Sturgis, H., "A Formalism for Translator Interactions", CACM 13, 10 (October 1970) pp 607-617

[Ev64] Evans, A. An Algol-60 Compiler, Annual Review of Automatic Programming, vol 4, Pergamon Press, 1964

[Fel64] Feldman, J., A Formal Semantics for Computer- oriented Languages, Ph.D. Dissertation, Carnegie- Mellon University, 1964

[FG68] Feldman, J. and Gries, D., "Translator Writing Systems", CACM 11, 2 (February 1968), pp 77-113

[FR69] Feldman, J. A. and Rovner, P. D. "An Algol-based Associative Language", CACM 12, 8 (August 1969), 439-449

[Fra70] Frailey, D. J., "Expression Optimization Using Unary Complement Operations", SIGPLAN Notices 5, 7 (July 1970), pp 67-85

[Ges72] Geschke, C. M., Global Program Optimizations, Ph.D. Dissertation, Carnegie-Mellon University, 1972

[Gim74] Gimpel, J. F., "Some Highlights of the SITBOL Language Extensions to SNOBOL4", SIGPLAN Notices 9, 10 (October 1974) pp 11-20

[Gris72] Griswold, R. E., The Macro Implementation of SNOBOL4, W. H. Freeman and Co., San Francisco, 1972

[Hew72] Hewitt, C., Description and Theoretical Analysis of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, Ph.D. Thesis, MIT, 1972

[Hop69] Hopgood, F. R. A., Compiling Techniques, American Elsevier, 1969

[IBM69] IBM System/360 Model 195: Functional Characteristics, IBM Publication No. S360-01/A22- 6943-0, August 1969

[John74] Johnsson, R. K., Thesis Proposal, October 1974

[Jon73] Jones, A. K., Protection in Programmed Systems, Ph.D. Dissertation, CMU, June 1973

[KKR65] Kanner, H., Kosinski, P., and Robinson, C. L., "The Structure of Yet Another Algol Compiler", in Programming Systems and Languages, S. Rosen, ed., McGraw-Hill, 1967

[Kn68] Knuth, D. E., The Art of Computer Programming: Volume 1: Fundamental Algorithms, Addison-Wesley Publishing Co., 1968

[LvdM73] Lindsey, C. H., and van der Meulen, S. G., Informal Introduction to ALGOL 68, American Elsevier, New York, 1973

[Mal72] Malcolm, M. A., "Algorithms to Reveal Properties of Floating-Point Arithmetic", CACM 15, 11 (November 1972) pp 949-951

[McK65] McKeeman, W. M., "Peephole Optimization", CACM 8, 7 (July 1965), pp 443-444

[Mil71] Miller, Perry L., Automatic Creation of a Code Generator from a Machine Description, Project MAC report no. TR-85, 1971

[MM75] Millstein, R. E. and Muntz, C. A., "The ILLIAC IV Compiler", SIGPLAN Notices 10, 3 (March 1975), pp 1-8

[OW69] Orgass, R. J. and Waite, W. M., "A Base for a Mobile Programming System", CACM 12, 9 (September 1969), pp 507-510

[Pain70] Painter, J. A., "Effectiveness of an Optimizing Compiler for Arithmetic Expressions", SIGPLAN Notices 5, 7 (July 1970) pp 101-126

[Par72] Parnas, D. L. "A Technique for Software Module Specification with Examples", CACM 15, 5 (May 1972), pp 330-336

[PJ75] Presberg, D. L. and Johnson, N. W., "The PARALYZER: IVTRAN's Parallelism Analyzer and Synthesizer", SIGPLAN Notices 10, 3 (March 1975), pp 9-16

[Rich71] Richards, M., "The Portability of the BCPL compiler", Software - Practice and Experience 1, (1971), pp 135-146

[Ros67] Rosen, S., "A Compiler-Building System Developed by Brooker and

Morris" in Programming Systems and Languages, S. Rosen, ed., McGraw-Hill, 1967

[Ryd72] Ryder, B. G., The FORTRAN verifier: Motivation and Implementation, Bell Laboratories, Murray Hill, N. J. (December 1972)

[Sim63] Simon, H. A., The Heuristic Compiler, Memorandum RM-3588-PR, The Rand Corporation, May 1963

[SM72] Sussman, G. and McDermott, D., "From PLANNER to CONNIVER: A Genetic Approach", FJCC 1972, pp 1171-1179

[Ste61] Steel, T. B., "A First Version of UNCOL", Proceedings of the Western Joint Computer Conference 1961, pp 371- 377, The Western Joint Computer Conference, 1961

[Str58] Strong, J., et al, "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM 1, 8 (August 1958) pp 12-18 and CACM 1, 9 (September 1958) pp 9-15

[SU70] Sethi, R. and Ullman, J., "The Generation of Optimal Code for Arithmetic Expressions", JACM 17, 4 (October 1970) pp 715-728

[VL73] VanLehn, K. A., (ed.) SAIL Users Manual, Stanford Artificial Intelligence Laboratory Operating Note 57.2, 1973

[vW69] van Wijngaarden, A. (ed), A Report on the Algorithmic Language Algol-68, Springer-Verlag, Berlin, 1969

[Wai69] Waite, W. M., "The STAGE-2 Macro Processor", Technical Report No. 69-3, Graduate School Computing Center, Boulder, Colorado, 1969

[Wai70] Waite, W. M., "The Mobile Programming System: STAGE- 2", CACM 13, 7 (July 1970), pp 415-421

[War74] Warren, J. C., Software Portability: A Summary of Related Concepts and Survey of Problems and Approaches, Technical Note No. 48, Digital Systems Laboratory, Stanford Un[illegible]rsity, September 1974

[WaS67] Warshall, S. and Shapiro, R. M., "A General-Purpose Table-Driven Compiler", in Programming Systems and Languages, S. Rosen, ed. McGraw-Hill 1967

[Weg72] Wegner, P., "The Vienna Definition Language", in Computing Surveys 4, 1 (March 1972) pp 5-63

[Wh73] White, John R. JOSSLE: A Language for Specifying and Structuring the Semantic Phase of Translators, Ph.D. Thesis, University of California Santa Barbara, June 1973

[Wir71] Wirth, N., "The programming language PASCAL", Acta Informatica 1, pp 35-63 (1971)

[Wu70] Wulf, W. A., et. al. BLISS: A Basic Language for Implementation of System Software for the PDP-10, Carnegie-Mellon University CSD report, 1970

[Wu71] Wulf, W.A., et. al. BLISS-11: Programmer's Manual, Digital Equipment Corp. 1972

[Wu73] Wulf, W. A. et. al. The Design of an Optimizing Compiler, Carnegie-Mellon University CSD report, 1973

[WU74] Wulf, W. A., et. al., ALPHARD: Toward a Language to Support Structured Programs, Carnegie-Mellon University CSD report, April 1974

[Wu74] Wulf, W. A., et. al., "HYDRA: The Kernel of a Multiprocessor Operating System", CACM 17, 6 (June 1974, pp 337-344

[Zw75] Zwakenberg, R. G. "Vector Extensions to LRLTRAN", SIGPLAN Notices 10, 3 (March 1975), pp 77-86

Citation cross-reference

Work	Citations	
AGG69	(1)	11
AI71	(1)	32
ASU70	(3)	43, 51, 106
Bai72	(1)	11
Bar73	(1)	34
Bea72	(2)	23, 72
BN71	(2)	34, 38
BS74	(1)	38
Buch74	(2)	102, 103
CH74	(1)	26
CLE69	(4)	7, 8, 16, 53
Cole74	(1)	15
Con58	(1)	13
CS70	(3)	15, 17, 42
Dah67	(2)	21, 32
DEC71	(1)	28
DEC72	(2)	82, 89
Don73	(5)	8, 9, 16, 17, 53
EN69	(3)	45, 75, 102
ER70	(1)	19
ES70	(1)	54
Ev64	(5)	7, 7, 15, 16, 16
Fel64	(4)	7, 7, 16, 16
FG68	(1)	7
FR69	(2)	21, 69
Fra70	(4)	19, 23, 48, 85
Ges72	(7)	15, 17, 28, 38, 42, 47, 108
Gim74	(1)	22
Gris72	(2)	13, 20
Hew72	(1)	60
Hop69	(2)	7, 48
IBM69	(1)	42
John74	(2)	31, 43
Jon73	(1)	33
KKR65	(2)	23, 85
Kn68	(2)	63, 86
LvdM73	(1)	67
Mal72	(1)	5
McK65	(1)	31
Mil71	(5)	8, 9, 16, 17, 53
MM75	(1)	42
OW69	(2)	12, 19
Pain70	(1)	25
Par72	(1)	5
PJ75	(1)	42
Rich71	(1)	14
Ros67	(3)	7, 8, 15

Ryd72	(1)	11
Sim63	(1)	102
SM72	(1)	60
Ste61	(3)	13, 13, 21
Str58	(2)	13, 13
SU70	(2)	43, 72
VL73	(1)	69
vW69	(3)	5, 5, 21
Wai69	(1)	20
Wai70	(2)	12, 15
War74	(1)	5
WaS67	(3)	7, 7, 15
Weg72	(1)	5
Wh73	(3)	7, 16, 16
Wir71	(1)	26
Wu70	(3)	28, 61, 85
Wu71	(2)	28, 43
Wu73	(9)	28, 28, 29, 31, 31, 35, 43, 47, 52
Wu74	(2)	32, 33
Zw75	(1)	42

This appendix gives a formal definition of the input language used by the implementation.

```
<expression> ::= <simple-expression> | <control-expression>

<simple-expression> ::= <logicalexp> ← <expression> | <logicalexp>

<logicalexp> ::= <relationalexp> | <logicalexp> <lop> <relationalexp>

<lop> ::= <logop> <attributes>

<logop> ::= AND | OR | XOR

<relationalexp> ::= <addexp> | <relationalexp> <rop> <addexp>

<rop> ::= <relop> <attributes>

<relop> ::= EQL | NEQ | LSS | GTR | GEQ | LEQ

<addexp> ::= <mulexp> | <addexp> <aop> <mulexp>

<aop> ::= <addop> <attributes>

<addop> ::= + | -

<mulexp> ::= <unexp> | <mulexp> <mop> <unexp>

<mop> ::= <mulop> <attributes>

<mulop> ::= * | / | MOD

<unexp> ::= <primary> | <uop> <unexp>

<uop> ::= <unop> <attributes>

<unop> ::= NOT | .

<primary> ::= <symbol> | (<expression>)

<symbol> ::= <number> <attributes> | <name> <attributes>

<control-expression> ::= <conditional> | <iterative>

<conditional> ::= IF <expression> THEN <expression> ELSE <expression>

<iterative> ::= WHILE <expression> DO <expression>

<attributes> ::= <empty> | [ <att-list> ]
```

<att-list> ::= <att-val-pair> | <att-val-pair> <att-list>

<att-val-pair> ::= <att-name> <att-values>

<att-values> ::= <att-val> | (<att-val-tuple>)

<att-val-tuple> ::= <att-val> | <att-val> <att-val-tuple>

Note that we do not define the following primitives; one may treat their definitions as the obvious intuitive ones:

<name> is an identifier in whatever characters are considered permissible.

<number> is a string of digits; we accept integers only but real numbers could be equally valid.

<att-name> is a member of the set $\mathcal{N}$ of attribute names (see page 56). Recall the convention that attribute names begin with a distinguished symbol, $ (see page 57).

<att-val> is a member of the set $\mathcal{V}$ of attribute values (page 56).

The output which follows represents an actual trace for one example, the expression .M1 + .M2. The output was generated by an extensive set of debugging aids developed by the author. The consequence is that some of the output uses conventions suitable for debugging rather than exposition. Some of these will be explained briefly, should anyone wish to study this trace in detail.

The appearance of a single quote (') associated with a node, operator, or identifier, indicates the presence of the attribute [$SIGN -] or [$SIGN OPPOSITE]. This was the shortest unambiguous way of providing this information in the trace.

The symbols PE001, ... , PEnnn represent trace depths, and allow the reader of the trace to associate a given level of trace with those which precede it, contain it, or are collateral to it. The search tree dump on pp 121-138 summarizes the search operations and provides an index into the trace.

```
LISP output 23:23 13-Apr-75 from core image SV642
PE001-------------------------------------------------------------------------------
Enter GENCODESET:   (( . M1 + . M2 ))
GENCODESET candidates
                           (1) (( . R + . EA ))
                                   (ADD R EA)
                                   (<COST> REF 2 SIZE 1)
                           (2) (( . R + . ' EA- ))
                                   (SUB R EA-)
                                   (<COST> REF 2 SIZE 1)
Merged GENCODESET candidates:
                           (1) (( . [ $SIGN (+ TRUE) $LOC REG $ADDR REG $POS 0 $SIZ 36 ] R [ $-
DT NO $SIGN (+ TRUE) $LOC LITERAL ] + . [ $LOC (REG MEM) $SIGN (- + TRUE) $ADDR (REG MEM S-
PREL) $POS 0 $SIZ 36 ] EA- [ $DT NO $SIGN (+ TRUE) $LOC EA ] ))
                                   <No code>
                                   NIL
                                                (ADD R EA)
                                                (<COST> REF 2 SIZE 1)
                                                (SUB R EA-)
                                                (<COST> REF 2 SIZE 1)
PE001-------------------------------------------------------------------------------
PE007--PE001------------------------------------------------------------------------
Enter PARALLEL-ENDORDER:
[+]--[.']--[EA-]
 I
 I
 [.]--[R]
[+]--[.]--[M2]
 I
 I
 [.]--[M1]
PE007--PE001--------------------------------------------------------------------------
PE008--PE007--PE001-------------------------------------------------------------------
Enter PARALLEL-ENDORDER:
[.]--[R]
[.]--[M1]
PE008--PE007--PE001-------------------------------------------------------------------
PE009--PE008--PE007--PE001------------------------------------------------------------
Enter PARALLEL-ENDORDER:
[R]
[M1]
PE009--PE008--PE007--PE001------------------------------------------------------------
PE009--PE008--PE007--PE001------------------------------------------------------------
Possible language axioms for:  (M1)
                         None
PE009--PE008--PE007--PE001------------------------------------------------------------
PE010--PE009--PE008--PE007--PE001-----------------------------------------------------
NODE-MATCH success:  (M1)=(R)
Sequence retained:
CODE                     COST
************************************************
PE013--PE009--PE008--PE007--PE001-----------------------------------------------------
PE008--PE007--PE001-------------------------------------------------------------------
Possible language axioms for:  (. M1)
                         None
PE008--PE007--PE001-------------------------------------------------------------------
PE011--PE008--PE007--PE001------------------------------------------------------------
NODE-MATCH failure:
                   <WANT> [$LOC REG $SIGN (+ TRUE) $ADDR REG $DT NIL $POS 0 $SIZ 36 ]
                   <HAVE> [$LOC MEM $SIGN (+ TRUE) $ADDR (MEM REG SPREL) $DT NIL $POS 0 $S→
IZ 36 ]
                   <DIFF> [$LOC M2R ]
                   (. M1) not = (. R)
PE011--PE008--PE007--PE001------------------------------------------------------------
PE012--PE011--PE008--PE007--PE001-----------------------------------------------------
GEN-ATTRIBUTE-XFORM possibilities:
                            (1) ($LOC M2R $SIGN SAME) ((MOVE R EA))
                            (2) ($LOC M2R $SIGN OPPOSITE) ((MOVN R EA))
PE012--PE011--PE008--PE007--PE001-----------------------------------------------------
PE013--PE012--PE011--PE008--PE007--PE001----------------------------------------------
Enter PARALLEL-ENDORDER:
[.]--[EA]
[.]--[M1]
PE013--PE012--PE011--PE008--PE007--PE001----------------------------------------------
PE014--PE013--PE012--PE011--PE008--PE007--PE001---------------------------------------
Enter PARALLEL-ENDORDER:
[EA]
[M1]
PE014--PE013--PE012--PE011--PE008--PE007--PE001---------------------------------------
PE013--PE012--PE011--PE008--PE007--PE001----------------------------------------------
Possible language axioms for:  (. M1)
                         None
PE013--PE012--PE011--PE008--PE007--PE001----------------------------------------------
PE015--PE013--PE012--PE011--PE008--PE007--PE001---------------------------------------
NODE-MATCH success:  (. M1)=(. EA)
Sequence retained:
CODE                     COST
************************************************
```

```
PE015--PE013--PE012--PE011--PE008--PE007--PE001------------------------------------------
PE012--PE011--PE008--PE007--PE001--------------------------------------------------------
GEN-ATTRIBUTE-XFORM success:  Satisfied by
                              (1) ($LOC M2R $SIGN SAME)
                                  ((MOVE R EA))
PE012--PE011--PE008--PE007--PE001--------------------------------------------------------
PE016--PE012--PE011--PE008--PE007--PE001-------------------------------------------------
Enter PARALLEL-ENDORDER:
[.]--[EA]
[.]--[M1]
PE016--PE012--PE011--PE008--PE007--PE001-------------------------------------------------
PE017--PE016--PE012--PE011--PE008--PE007--PE001------------------------------------------
Enter PARALLEL-ENDORDER:
[EA]
[M1]
PE017--PE016--PE012--PE011--PE008--PE007--PE001------------------------------------------
PE016--PE012--PE011--PE008--PE007--PE001-------------------------------------------------
Possible language axioms for:  (. M1)
                              None
PE016--PE012--PE011--PE008--PE007--PE001-------------------------------------------------
PE018--PE016--PE012--PE011--PE008--PE007--PE001------------------------------------------
NODE-MATCH success:  (. M1)=(. EA)
Sequence retained:
CODE                          COST
**********************************************
PE018--PE016--PE012--PE011--PE008--PE007--PE001------------------------------------------
PE012--PE011--PE008--PE007--PE001--------------------------------------------------------
GEN-ATTRIBUTE-XFORM success:  Satisfied by
                              (2) ($LOC M2R $SIGN OPPOSITE)
                                  ((MOVN R EA))
PE012--PEU11--PE008--PE007--PE001--------------------------------------------------------
PE019--PE007--PE001----------------------------------------------------------------------
Enter PARALLEL-ENDORDER:
[.']--[EA-]
[.]--[M2]
PE019--PE007--PE001----------------------------------------------------------------------
PE020--PE019--PE007--PE001---------------------------------------------------------------
Enter PARALLEL-ENDORDER:
[EA-]
[M2]
PE020--PE019--PE007--PE001---------------------------------------------------------------
PE020--PE019--PE007--PE001---------------------------------------------------------------
Possible language axioms for:  (M2)
                              None
PE020--PE019--PE007--PE001---------------------------------------------------------------
PE021--PE020--PE019--PE007--PE001--------------------------------------------------------
NODE-MATCH failure:
                        <WANT> [$LOC EA $SIGN (+ TRUE) $ADDR NIL $DT ND $POS NIL $SIZ NIL ]
                        <HAVE> [$LOC LITERAL $SIGN (+ TRUE) $ADDR LITERAL $DT NJ $POS 8 $SIZ 18→

                        <DIFF> [$LOC L2EA ]
                        (M2) not = (EA-)
PE021--PE020--PE019--PE007--PE001--------------------------------------------------------
PE022--PE021--PE020--PE019--PE007--PE001-------------------------------------------------
GEN-ATTRIBUTE-XFORM possibilities:
                              (1) ($LOC L2EA $SIGN SAME) NIL
PE022--PE021--PE020--PE019--PE007--PE001-------------------------------------------------
PE023--PE022--PE021--PE020--PE019--PE007--PE001------------------------------------------
Enter PARALLEL-ENDORDER:
```

```
[X]
[M2]
PE023--PE022--PE021--PE020--PE019--PE007--PE001------------------------------------------
PE022--PE021--PE020--PE019--PE007--PE001-------------------------------------------------
GEN-ATTRIBUTE-XFORM success:  Satisfied by
                               (1) ($LOC L2EA $SIGN SAME)
                                  <No code>
PE022--PE021--PE020--PE019--PE007--PE001-------------------------------------------------
PE019--PE007--PE001----------------------------------------------------------------------
Possible language axioms for:  (. M2)
                        None
PE019--PE007--PE001----------------------------------------------------------------------
PE024--PE019--PE007--PE001---------------------------------------------------------------
NODE-MATCH success:  (. M2)=(. ' EA-)
Sequence retained:
CODE                     COST
*************************************************
  <T-op>            REF 0 SIZE 0        (M2)
         Total      REF 0 SIZE 0
PE024--PE019--PE007--PE001---------------------------------------------------------------
PE007--PE001-----------------------------------------------------------------------------
Possible language axioms for:  (. M1 + . M2)
+                  ( S1 I $SUBTREE T $SIGN OPPOSITE ) + [ $SIGN OPPOSITE ] S2 [ $SUBTREE
                      T $SIGN OPPOSITE ) )
                 :=
                   ( S1 I $SUBTREE T ) + S2 [ $SUBTREE T ] )
                   [:=]--[+]--[S2]
                    I      I
                    I      [S1]
                    I
                    [+']--[S2']
                    I
                    [S1']
                   Pre: T
                   Post: NIL
*************************************************
PE007--PE001-----------------------------------------------------------------------------
PE028--PE007--PE001----------------------------------------------------------------------
NODE-MATCH success:  (. M1 + . M2)=(. R + . ' EA-)
Sequence retained:
CODE                     COST
*************************************************
(MOVE R EA)         REF 2 SIZE 1        (. M1)
  <T-op>            REF 0 SIZE 0        (M2)
         Total      REF 2 SIZE 1
PE028--PE007--PE001----------------------------------------------------------------------
PE029--PE007--PE001----------------------------------------------------------------------
NODE-MATCH failure (SUBNODE mismatch):
LD:
  <WANT> [$LOC REG $SIGN (- COMP) $ADDR (MEM REG SPREL) $OT NIL $POS 0 $SIZ 36 ]
  <HAVE> [$LOC REG $SIGN (+ TRUE) $ADDR REG $OT NIL $POS 0 $SIZ 36 ]
  <DIFF> [$SIGN OPPOSITE ]
(. ' M1 + ' . ' M2) not = (. R + . ' EA-)
Sequence rejected:
CODE                     COST
*************************************************
(MOVE R EA)         REF 2 SIZE 1        (. ' M1)
  <T-op>            REF 0 SIZE 0        (M2)
         Total      REF 2 SIZE 1
```

```
PE029--PED07--PED01------------------------------------------------------------------
PE007--PE001-------------------------------------------------------------------------
Possible language axioms for:  (. ' M1 + . M2)
+                  ( S1 [ $SUBTREE T $SIGN OPPOSITE ] + I $SIGN OPPOSITE ] S2 I $SUBTREE
                     T $SIGN OPPOSITE ] )
                 :=
                   ( S1 [ $SUBTREE T ] + S2 I $SUBTREE T ] )
                   [:=]--[+]--[S2]
                    I     I
                    I     [S1]
                    I
                    [+']--[S2']
                    I
                    [S1']
                   Pre: T
                   Post: NIL
*********************************************
PE007--PE001-------------------------------------------------------------------------
PE033--PED07--PED01------------------------------------------------------------------
NODE-MATCH failure (SUBNODE mismatch):
LO:
  <WANT> [$LOC REG $SIGN (- COMP) $ADDR (REG MEM) $DT NIL $POS 0 $SIZ 36 ]
  <HAVE> [$LOC REG $SIGN (+ TRUE) $ADDR REG $DT NIL $POS 0 $SIZ 36 ]
  <DIFF> [$SIGN OPPOSITE ]
(. ' M1 + . M2) not = (. R + . ' EA-)
Sequence rejected:
CODE                     COST
*********************************************
(MOVN R EA)        REF 2 SIZE 1        (. ' M1)
  <T-op>           REF 0 SIZE 0        (M2)
        Total      REF 2 SIZE 1
PE033--PE007--PED01------------------------------------------------------------------
PE034--PE007--PE001------------------------------------------------------------------
NODE-MATCH success:  (. M1 + ' . ' M2)=(. R + . ' EA-)
Sequence retained:
CODE                     COST
*********************************************
(MOVN R EA)        REF 2 SIZE 1        (. M1)
  <T-op>           REF 0 SIZE 0        (M2)
        Total      REF 2 SIZE 1
PE034--PE007--PE001------------------------------------------------------------------
PE001--------------------------------------------------------------------------------
GENCODESET possibility:  (1)
(( (( . M1 + . M2 )) (( . M1 + ' . ' M2 )) ))
(*NODESET
 (*TREE
  (*EXPR
   (*LO
    (*TREE
     (*EXPR
      (*LO
       (*TREE
        (*EXPR
         (*ATOM M1)
         (*OP ATF)
         (*ATTRIB
          (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
      (*OP /.)
      (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (+ TRUE) $LOC REG)))
     (*MACHINE-CODE
      (*MACHOP
       (*TYPE ($LOC M2R $SIGN SAME))
       (*OP
        (*TREE
         (*EXPR
          (*OP /.)
          (*LO
           (*TREE
            (*EXPR (*ATOM EA)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
          (*ATTRIB (<UF-ATT> $SIGN (TRUE +) $LOC MEM)))))
       (*RESULT
        (*TREE
         (*EXPR
          (*OP /.)
          (*LO
           (*TREE
            (*EXPR (*ATOM R)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
          (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG)))))
       (*SE ((EQUAL (VALUE /.R) (VALUE (/. EA)))))
       (*CODE ((MOVE R EA)))
       (*COST (<COST> REF 2 SIZE 1))
       (*LABEL ($LOC MEM -> $LOC REG))))))
   (*RO
    (*TREE
     (*EXPR
      (*LO
       (*TREE
        (*EXPR
         (*ATOM M2)
         (*OP ATF)
         (*ATTRIB (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (TRUE +) $LOC EA)))
        (*MACHINE-CODE
         (*MACHOP
          (*TYPE ($LOC L2EA $SIGN SAME))
          (*OP
           (*TREE
            (*EXPR (*ATOM X)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
          (*RESULT
           (*TREE
            (*EXPR (*ATOM EA)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
          (*SE (NIL))
          (*CODE NIL)
          (*COST (<COST> REF 0 SIZE 0))
          (*LABEL ($LOC LITERAL -> $LOC EA))))))
      (*OP /.)
      (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (+ TRUE) $LOC MEM))))
   (*OP +)
   (*ATTRIB (<BF-ATT>))))
 (*TREE
  (*EXPR
(*OP +)
(*LO
 (*TREE
  (*EXPR
   (*LO
    (*TREE
     (*EXPR
      (*ATOM M1)
      (*OP ATF)
      (*ATTRIB
       (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
   (*OP /.)
   (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (REG MEM) $SIGN (+ TRUE) $LOC REG)))
  (*MACHINE-CODE
   (*MACHOP
    (*TYPE ($LOC M2R $SIGN OPPOSITE))
    (*OP
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM EA)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC LITERAL)))))
       (*ATTRIB (<UF-ATT> $SIGN NIL $LOC MEM)))))
    (*RESULT
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM R)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*ATTRIB
        (<UF-ATT> $SIGN
                  (LAMBDA (FROM TO) (CHANGE-SIGN FROM TO))
                  $LOC
                  REG
                  $ADDR
                  (REG MEM))))))
     (*SE ((EQUAL (VALUE /.R) (VALUE (- (/. EA))))))
     (*CODE ((MOVN R EA)))
     (*COST (<COST> REF 2 SIZE 1))
     (*LABEL ($LOC MEM -> $LOC REG $SIGN OPPOSITE))))))
 (*RO
  (*TREE
   (*EXPR
    (*LO
     (*TREE
      (*EXPR
       (*ATOM M2)
       (*OP ATF)
       (*ATTRIB (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (TRUE +) $LOC EA)))
      (*MACHINE-CODE
       (*MACHOP
        (*TYPE ($LOC L2EA $SIGN SAME))
        (*OP
          (*TREE
           (*EXPR (*ATOM X)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
          (*RESULT
           (*TREE
            (*EXPR (*ATOM EA)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
          (*SE (NIL))
          (*CODE NIL)
          (*COST (<COST> REF 0 SIZE 0))
          (*LABEL ($LOC LITERAL -> $LOC EA))))))
      (*OP /.)
      (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (- COMP) $LOC MEM)))))
   (*ATTRIB (<BF-ATT> $SIGN (- COMP))))))
CODE                      COST
**************************************************
(MOVE R EA)        REF 2 SIZE 1        (. M1)
  <T-op>           REF 0 SIZE 0        (M2)
        Total      REF 2 SIZE 1
**************************************************
(MOVN R EA)        REF 2 SIZE 1        (. M1)
  <T-op>           REF 0 SIZE 0        (M2)
        Total      REF 2 SIZE 1
PE001------------------------------------------------------------------------------
(*CODESET
 (*NODESET
  (*TREE
   (*EXPR
    (*LO
     (*TREE
      (*EXPR
       (*LO
        (*TREE
         (*EXPR
          (*ATOM M1)
          (*OP ATF)
          (*ATTRIB
           (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*OP /.)
       (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (+ TRUE) $LOC REG)))
      (*MACHINE-CODE
       (*MACHOP
        (*TYPE ($LOC M2R $SIGN SAME))
        (*OP
         (*TREE
          (*EXPR
           (*OP /.)
           (*LO
            (*TREE
             (*EXPR (*ATOM EA)
                    (*OP ATF)
                    (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
           (*ATTRIB (<UF-ATT> $SIGN (TRUE +) $LOC MEM)))))
        (*RESULT
         (*TREE
          (*EXPR
           (*OP /.)
         (*LO
          (*TREE
           (*EXPR (*ATOM R)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
         (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG)))))
      (*SE ((EQUAL (VALUE /.R) (VALUE (/. EA)))))
      (*CODE ((MOVE R EA)))
      (*COST (<COST> REF 2 SIZE 1))
      (*LABEL ($LOC MEM -> $LOC REG))))))
  (*RO
   (*TREE
    (*EXPR
     (*LO
      (*TREE
       (*EXPR
        (*ATOM M2)
        (*OP ATF)
        (*ATTRIB
         (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (TRUE +) $LOC EA)))
       (*MACHINE-CODE
        (*MACHOP
         (*TYPE ($LOC L2EA $SIGN SAME))
         (*OP
          (*TREE
           (*EXPR (*ATOM X)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
         (*RESULT
          (*TREE
           (*EXPR (*ATOM EA)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
         (*SE (NIL))
         (*CODE NIL)
         (*COST (<COST> REF 0 SIZE 0))
         (*LABEL ($LOC LITERAL -> $LOC EA))))))
    (*OP /.)
    (*ATTRIB
     (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (+ TRUE) $LOC MEM)))))
 (*OP +)
 (*ATTRIB (<BF-ATT> $SIZ 36 $POS 0 $LOC REG $SIGN +)))
(*MACHINE-CODE
 (*MACHOP
  (*TYPE +)
  (*OP
   (*TREE
    (*EXPR
     (*OP +)
     (*LO
      (*TREE
       (*EXPR
        (*OP /.)
        (*LO
         (*TREE
          (*EXPR (*ATOM R)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
        (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG $ADDR REG $POS 0 $SIZ 36)))))
       (*RO
        (*TREE
         (*EXPR
          (*OP /.)
          (*LO
           (*TREE
            (*EXPR (*ATOM EA)
                   (*OP ATF)
                   (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC EA)))))
          (*ATTRIB
           (<UF-ATT> $SIGN
                     (+ TRUE)
                     $LOC
                     (REG MEM)
                     $ADDR
                     (REG MEM SPREL)
                     $POS
                     0
                     $SIZ
                     36)))))
       (*ATTRIB (<BF-ATT>)))))
    (*RESULT
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM R)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*ATTRIB (<UF-ATT> $SIGN + $LOC REG $POS 0 $SIZ 36)))))
    (*SE ((EQUAL (VALUE (/. R)) (+ (VALUE (/. R)) (VALUE (/. EA))))))
    (*CODE ((ADD R EA)))
    (*COST (<COST> REF 2 SIZE 1))
    (*LABEL (Add R,EA)))))
 (*TREE
  (*EXPR
   (*OP +)
   (*LO
    (*TREE
     (*EXPR
      (*LO
       (*TREE
        (*EXPR
         (*ATOM M1)
         (*OP ATF)
         (*ATTRIB
          (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
      (*OP /.)
      (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (REG MEM) $SIGN (+ TRUE) $LOC REG)))
     (*MACHINE-CODE
      (*MACHOP
       (*TYPE ($LOC M2R $SIGN OPPOSITE))
       (*OP
        (*TREE
         (*EXPR
          (*OP /.)
          (*LO
           (*TREE
          (*EXPR (*ATOM EA)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC LITERAL)))))
        (*ATTRIB (<UF-ATT> $SIGN NIL $LOC MEM)))))
     (*RESULT
      (*TREE
       (*EXPR
        (*OP /.)
        (*LO
         (*TREE
          (*EXPR (*ATOM R)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
        (*ATTRIB
         (<UF-ATT> $SIGN
                   (LAMBDA (FROM TO) (CHANGE-SIGN FROM TO))
                   $LOC
                   REG
                   $ADDR
                   (REG MEM))))))
     (*SE ((EQUAL (VALUE /.R) (VALUE (- (/. EA))))))
     (*CODE ((MOVN R EA)))
     (*COST (<COST> REF 2 SIZE 1))
     (*LABEL ($LOC MEM -> $LOC REG $SIGN OPPOSITE))))))
 (*RO
  (*TREE
   (*EXPR
    (*LO
     (*TREE
      (*EXPR
       (*ATOM M2)
       (*OP ATF)
       (*ATTRIB
        (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (TRUE +) $LOC EA)))
      (*MACHINE-CODE
       (*MACHOP
        (*TYPE ($LOC L2EA $SIGN SAME))
        (*OP
         (*TREE
          (*EXPR (*ATOM X)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
        (*RESULT
         (*TREE
          (*EXPR (*ATOM EA)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
        (*SE (NIL))
        (*CODE NIL)
        (*COST (<COST> REF 3 SIZE 0))
        (*LABEL ($LOC LITERAL -> $LOC EA))))))
    (*OP /.)
    (*ATTRIB
     (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (- COMP) $LOC MEM)))))
 (*ATTRIB (<BF-ATT> $SIZ 36 $POS 0 $LOC REG $SIGN +)))
(*MACHINE-CODE
 (*MACHOP
  (*TYPE +)
  (*OP
   (*TREE
    (*EXPR
     (*OP +)
     (*LO
      (*TREE
       (*EXPR
        (*OP /.)
        (*LO
         (*TREE
          (*EXPR (*ATOM R)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
        (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG $ADDR REG $POS 0 $SIZ 36)))))
     (*RO
      (*TREE
       (*EXPR
        (*OP /.)
        (*LO
         (*TREE
          (*EXPR (*ATOM EA)
                 (*OP ATF)
                 (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC EA)))))
        (*ATTRIB
         (<UF-ATT> $SIGN
                   (+ TRUE)
                   $LOC
                   (REG MEM)
                   $ADDR
                   (REG MEM SPREL)
                   $POS
                   0
                   $SIZ
                   36)))))
      (*ATTRIB (<BF-ATT>)))))
   (*RESULT
    (*TREE
     (*EXPR
      (*OP /.)
      (*LO
       (*TREE
        (*EXPR (*ATOM R)
               (*OP ATF)
               (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
      (*ATTRIB (<UF-ATT> $SIGN + $LOC REG $POS 0 $SIZ 36)))))
   (*SE ((EQUAL (VALUE (/. R)) (+ (VALUE (/. R)) ([illegible]UE (/. EA))))))
   (*CODE ((ADD R EA)))
   (*COST (<COST> REF 2 SIZE 1))
   (*LABEL (Add R,EA)))))
(*TREE
 (*EXPR
  (*LO
   (*TREE
    (*EXPR
     (*LO
      (*TREE
       (*EXPR
        (*ATOM M1)
        (*OP ATF)
        (*ATTRIB
      (<ATF-ATT> $SIZ 18 $POS 0 $AOOR LITERAL $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
   (*OP /.)
   (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $AOOR (MEM REG SPREL) $SIGN (+ TRUE) $LOC REG)))
  (*MACHINE-COOE
   (*MACHOP
    (*TYPE ($LOC M2R $SIGN SAME))
    (*OP
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM EA)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $OT NO $SIGN (TRUE +) $LOC EA)))))
       (*ATTRIB (<UF-ATT> $SIGN (TRUE +) $LOC MEM)))))
    (*RESULT
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM R)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG)))))
    (*SE ((EQUAL (VALUE /.R) (VALUE (/. EA)))))
    (*COOE ((MOVE R EA)))
    (*COST (<COST> REF 2 SIZE 1))
    (*LABEL ($LOC MEM -> $LOC REG))))))
(*RO
 (*TREE
  (*EXPR
   (*LO
    (*TREE
     (*EXPR
      (*ATOM M2)
      (*OP ATF)
      (*ATTRIB
       (<ATF-ATT> $SIZ 18 $POS 0 $AOOR LITERAL $OT NO $SIGN (TRUE +) $LOC EA)))
     (*MACHINE-COOE
      (*MACHOP
       (*TYPE ($LOC L2EA $SIGN SAME))
       (*OP
        (*TREE
         (*EXPR (*ATOM X)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*RESULT
        (*TREE
         (*EXPR (*ATOM EA)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $OT NO $SIGN (TRUE +) $LOC EA)))))
       (*SE (NIL))
       (*COOE NIL)
       (*COST (<COST> REF 0 SIZE 0))
       (*LABEL ($LOC LITERAL -> $LOC EA))))))
   (*OP /.)
   (*ATTRIB
      (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (+ TRUE) $LOC MEM)))))
  (*OP +)
  (*ATTRIB (<BF-ATT> $SIZ 36 $POS 0 $LOC REG $SIGN +)))
 (*MACHINE-CODE
  (*MACHOP
   (*TYPE +)
   (*OP
    (*TREE
     (*EXPR
      (*OP +)
      (*LO
       (*TREE
        (*EXPR
         (*OP /.)
         (*LO
          (*TREE
           (*EXPR (*ATOM R)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
         (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG $ADDR REG $POS 0 $SIZ 36)))))
      (*RO
       (*TREE
        (*EXPR
         (*OP /.)
         (*LO
          (*TREE
           (*EXPR (*ATOM EA-)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC EA)))))
         (*ATTRIB
          (<UF-ATT> $SIGN - $LOC (REG MEM) $ADDR (REG MEM SPREL) $POS 0 $SIZ 36)))))
      (*ATTRIB (<BF-ATT>)))))
   (*RESULT
    (*TREE
     (*EXPR
      (*OP /.)
      (*LO
       (*TREE
        (*EXPR (*ATOM R)
               (*OP ATF)
               (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
      (*ATTRIB (<UF-ATT> $SIGN + $LOC REG $POS 0 $SIZ 36)))))
   (*SE ((EQUAL (VALUE (/. R)) (+ (VALUE (/. R)) (VALUE (/. EA-))))))
   (*CODE ((SUB R EA-)))
   (*COST (<COST> REF 2 SIZE 1))
   (*LABEL (Sub R,EA-)))))
(*TREE
 (*EXPR
  (*OP +)
  (*LO
   (*TREE
    (*EXPR
     (*LO
      (*TREE
       (*EXPR
        (*ATOM M1)
        (*OP ATF)
        (*ATTRIB
         (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
   (*OP /.)
   (*ATTRIB (<UF-ATT> $SIZ 36 $POS 0 $ADDR (REG MEM) $SIGN (+ TRUE) $LOC REG)))
  (*MACHINE-CODE
   (*MACHOP
    (*TYPE ($LOC M2R $SIGN OPPOSITE))
    (*OP
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM EA)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC LITERAL)))))
       (*ATTRIB (<UF-ATT> $SIGN NIL $LOC MEM)))))
    (*RESULT
     (*TREE
      (*EXPR
       (*OP /.)
       (*LO
        (*TREE
         (*EXPR (*ATOM R)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*ATTRIB
        (<UF-ATT> $SIGN
                  (LAMBDA (FROM TO) (CHANGE-SIGN FROM TO))
                  $LOC
                  REG
                  $ADDR
                  (REG MEM))))))
    (*SE ((EQUAL (VALUE /.R) (VALUE (- (/. EA))))))
    (*CODE ((MOVN R EA)))
    (*COST (<COST> REF 2 SIZE 1))
    (*LABEL ($LOC MEM -> $LOC REG $SIGN OPPOSITE))))))
(*RO
 (*TREE
  (*EXPR
   (*LO
    (*TREE
     (*EXPR
      (*ATOM M2)
      (*OP ATF)
      (*ATTRIB
       (<ATF-ATT> $SIZ 18 $POS 0 $ADDR LITERAL $DT NO $SIGN (TRUE +) $LOC EA)))
     (*MACHINE-CODE
      (*MACHOP
       (*TYPE ($LOC L2EA $SIGN SAME))
       (*OP
        (*TREE
         (*EXPR (*ATOM X)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (+ TRUE) $LOC LITERAL)))))
       (*RESULT
        (*TREE
         (*EXPR (*ATOM EA)
                (*OP ATF)
                (*ATTRIB (<ATF-ATT> $DT NO $SIGN (TRUE +) $LOC EA)))))
       (*SE (NIL))
```

```
            (*CODE NIL)
            (*COST (<COST> REF 0 SIZE 0))
            (*LABEL ($LOC LITERAL -> $LOC EA))))))
        (*OP /.)
        (*ATTRIB
         (<UF-ATT> $SIZ 36 $POS 0 $ADDR (MEM REG SPREL) $SIGN (- COMP) $LOC MEM)))))
     (*ATTRIB (<BF-ATT> $SIZ 36 $POS 0 $LOC REG $SIGN +)))
    (*MACHINE-CODE
     (*MACHOP
      (*TYPE +)
      (*OP
       (*TREE
        (*EXPR
         (*OP +)
         (*LO
          (*TREE
           (*EXPR
            (*OP /.)
            (*LO
             (*TREE
              (*EXPR (*ATOM R)
                     (*OP ATF)
                     (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
            (*ATTRIB (<UF-ATT> $SIGN (+ TRUE) $LOC REG $ADDR REG $POS 0 $SIZ 36)))))
         (*RO
          (*TREE
           (*EXPR
            (*OP /.)
            (*LO
             (*TREE
              (*EXPR (*ATOM EA-)
                     (*OP ATF)
                     (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC EA)))))
            (*ATTRIB
             (<UF-ATT> $SIGN - $LOC (REG MEM) $ADDR (REG MEM SPREL) $POS 0 $SIZ 36)))))
         (*ATTRIB (<BF-ATT>)))))
      (*RESULT
       (*TREE
        (*EXPR
         (*OP /.)
         (*LO
          (*TREE
           (*EXPR (*ATOM R)
                  (*OP ATF)
                  (*ATTRIB (<ATF-ATT> $OT NO $SIGN (+ TRUE) $LOC LITERAL)))))
         (*ATTRIB (<UF-ATT> $SIGN + $LOC REG $POS 0 $SIZ 36)))))
      (*SE ((EQUAL (VALUE (/. R)) (+ (VALUE (/. R)) (VALUE (/. EA-))))))
      (*CODE ((SUB R EA-)))
      (*COST (<COST> REF 2 SIZE 1))
      (*LABEL (Sub R,EA-)))))))
(178200 msec 21295 CONSes 7950 GC-time 170250 non-GC-time 4.4612794 % GC-time)
PE001------------------------------------------------------------------------------------
CODE                     COST
*************************************************
(MOVE R EA)        REF 2 SIZE 1        (. M1)
  <T-op>           REF 0 SIZE 0        (M2)
(ADD R EA)         REF 2 SIZE 1        (. M1 + . M2)
         Total     REF 4 SIZE 2
*************************************************
```

```
(MOVN R EA)          REF 2 SIZE 1          (. M1)
  <T-op>             REF 0 SIZE 0          (M2)
(ADD R EA)           REF 2 SIZE 1          (. M1 + . ' M2)
         Total       REF 4 SIZE 2
**************************************************
(MOVE R EA)          REF 2 SIZE 1          (. M1)
  <T-op>             REF 0 SIZE 0          (M2)
(SUB R EA-)          REF 2 SIZE 1          (. M1 + . M2)
         Total       REF 4 SIZE 2
**************************************************
(MOVN R EA)          REF 2 SIZE 1          (. M1)
  <T-op>             REF 0 SIZE 0          (M2)
(SUB R EA-)          REF 2 SIZE 1          (. M1 + . ' M2)
         Total       REF 4 SIZE 2
PE001-------------------------------------------------------------------------------
Search tree dump  23:31 13-Apr-75
------------------------------------------------------------------------------------
PE001 Test case: (. M1 + . M2)
------------------------------------------------------------------------------------
       PE007 Comparing (. R + . ' EA-) and (. M1 + . M2)
       -----------------------------------------------------------------------------
              PE008 Comparing (. R) and (. M1)
              ----------------------------------------------------------------------
                     PE009 Comparing (R) and (M1)
                     ---------------------------------------------------------------
                            PE010 Iterate across axiom transformation: (M1)
                            --------------------------------------------------------
                     PE011 Iterate across axiom transformation: (. M1)
                     ---------------------------------------------------------------
                            PE012 Attribute transformation: need <DIFF> $LOC M2R
                            --------------------------------------------------------
                                   PE013 Comparing (. EA) and (. M1)
                                   -------------------------------------------------
                                          PE014 Comparing (EA) and (M1)
                                          ------------------------------------------
                                          PE015 Iterate across axiom transformation: (. M1+
)

                                          ------------------------------------------
                                   PE016 Comparing (. EA) and (. M1)
                                   -------------------------------------------------
                                          PE017 Comparing (EA) and (M1)
                                          ------------------------------------------
                                          PE018 Iterate across axiom transformation: (. M1+
)

                                          ------------------------------------------
              PE019 Comparing (. ' EA-) and (. M2)
              ----------------------------------------------------------------------
                     PE020 Comparing (EA-) and (M2)
                     ---------------------------------------------------------------
                            PE021 Iterate across axiom transformation: (M2)
                            --------------------------------------------------------
                                   PE022 Attribute transformation: need <DIFF> $LOC L2EA
                                   -------------------------------------------------
                                          PE023 Comparing (X) and (M2)
                                          ------------------------------------------
                     PE024 Iterate across axiom transformation: (. M2)
                     ---------------------------------------------------------------
              PE028 Iterate across axiom transformation: (. M1 + . M2)
              ----------------------------------------------------------------------
```

```
PE029 Iterate across axiom transformation: (. ' M1 + ' . ' M2)
------------------------------------------------------------------------------
PE033 Iterate across axiom transformation: (. ' M1 + . M2)
------------------------------------------------------------------------------
PE034 Iterate across axiom transformation: (. M1 + ' . ' M2)
------------------------------------------------------------------------------
```

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 18 AFOSR - TR - 75 - 1679	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 MACHINE INDEPENDENT GENERATION OF OPTIMAL LOCAL CODE,	5. TYPE OF REPORT & PERIOD COVERED Interim
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 Joseph M. Newcomer	8. CONTRACT OR GRANT NUMBER(s) 15 F44620-73-C-0074, ARPA Order-2466
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Dept. Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101E AO 2466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209	12. REPORT DATE 11 May 1975
	13. NUMBER OF PAGES 143
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research Bolling AFB, DC 20332 12 146p.	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

9 Doctoral thesis,

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

There has been extensive research into the automatic generation of compilers. Much of this has concentrated on the issues of syntax and semantics, while little has been done on the problems of code generation. This thesis represents one approach to the latter problem. A model of a compiler-compiler is presented, with the research focussing on the construction of one component of the compiler, that module which determines the possible code sequences which realize a given program. The input to this component is a set of code sequences which are possible realizations of each language construct. This thesis concentrates on the automatic

CONTINUED

over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 403 081 UNCLASSIFIED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

CONTINUED 20. abstract

generation of these code sequences from a formal description of the hardware and the language. A notation is developed for representing maching instructions, and a prototype system has been constructed to demonstrate that this notation is amenable to automated analysis.

UNCLASSIFIED